BLAZEPHOENIX PROTOCOL · COMPLETE TECHNICAL WHITEPAPER

BlazePhoenix

A universal liquidity-aggregation engine and a provably-solvent staking engine: one stateless mathematical core across the entire protocol — on-chain routing, on-chain yield, on-chain solvency, zero off-chain trust. Stateless. Serverless. Ungovernable.

STATELESS · SERVERLESS · UNGOVERNABLE — WEB 3.0, AS SPECIFIED

$$\mathcal{U}(\text{route}) \;=\; \mathbb{1}\big[\,y_{\text{net}} \geq \Phi \cdot y_{\text{quote}}\,\big] \cdot y_{\text{net}} \cdot \prod_i \Psi_i^{\,w_i}$$

Version 1.0.0
June 2026
Ethereum · Base · Arbitrum · Optimism
License: BUSL-1.1
Anonymous Mitra

"Every other contract imports the Core. No math is redefined anywhere else, and nothing trusts a number it did not compute."

ABSTRACT

We present **BlazePhoenix** — a complete, on-chain financial protocol whose two interlocking engines, an *aggregator* and a *staking vault*, share one mathematical discipline: *compute, do not trust*. Every price, every yield, every solvency proof is derived from chain state by pure arithmetic. No off-chain solver. No price oracle. No keeper. No administrator capable of touching user funds. The chain is not a settlement rail for decisions made elsewhere; it is where every decision is made.

Part I — The Aggregator (§1–§18). A universal liquidity-aggregation engine that reduces every AMM to one of a small family of closed-form output functions. Four universal operators — the **Eightfold Dispatcher $\mathbf{\Omega}$**, the **Deterministic Derivation $\mathscr{D}$**, the **Iron-Law floor $\mathbf{\Phi}$**, and the **Vitality Field $\mathbf{\Psi}$** — collapse aggregation cost from linear in venues to constant in shapes. The **Monoslot** encodes a pool's entire state in one 256-bit word, read in a single SLOAD. A **self-healing registry** learns liquidity by trading it, with no keeper and no indexer. A **capital-anchored split allocator** defeats dust-pool median manipulation because truth votes with capital, not existence. The **Exact Pass** — revert-extraction from the pool's own bytecode — guarantees that the number shown off-chain equals the number realised on-chain. Validated across Ethereum and three Layer-2 networks from one identical bytecode: ~38% warm-cache gas saving, quote–execution agreement to within the protocol fee.

Part II — The Staking Engine (§19–§26). A provably-solvent, autonomously-maintained single-asset vault for BZPX. Because collateral and borrowed asset are the same token, *no price oracle exists anywhere* — an entire attack surface is absent by design. The protocol stands behind one equation, the **Master Conservation Identity**, enforced per-transaction by the `conserves` modifier: any transaction that would leave the ledger claiming more than the contract holds reverts atomically. Solvency is not a dashboard reading; it is a revert condition. A quadratic boost curve rewards lock commitment up to **2.75×** at the seven-year horizon. Two disjoint MasterChef-style accumulators pay protocol emission to borrowers and borrower interest to pure savers with zero cross-subsidy — the **Dual-Accumulator Doctrine**. **Autonomous Maintenance** — carried by every ordinary user transaction, gas-bounded, keeper-free — liquidates the underwater and refreshes positions without a cron job or off-chain bot. The **permissionless circuit-breaker** can be tripped by anyone when the blockchain itself proves insolvency, and reversed by the admin when it does not. There is no backdoor sweep; reserve withdrawals flow only to an immutable treasury address.

> Together, the two engines implement the BlazePhoenix thesis: a protocol that is **stateless** at its Core, **serverless** in its operation, and **ungovernable** in the dimensions where governance is a weapon — Web 3.0 as specified, not as marketed.

Anonymous Mitra · June 2026 · Version 1.0.0 · BUSL-1.1

Contents

PART I · THE AGGREGATOR

1 Introduction

A trader who wants the best price for a swap faces a paradox. The liquidity exists — spread across dozens of decentralized exchanges, each with its own pool contracts, its own pricing curve, its own address scheme. But no single venue holds it all. The price of finding the best execution is the cost of speaking every exchange's language at once, and of trusting every translator in the chain between the price you were shown and the trade that finally lands.

The incumbent answer is to integrate venues one at a time: a connector for Uniswap, another for Curve, another for Balancer, each with bespoke quoting code, bespoke address lookups, bespoke execution paths. The integration surface grows linearly with the number of venues, and every new connector is a fresh opportunity for a pricing bug, a stale assumption, or a silent mismatch between what was quoted and what executes. Worse, the industry's dominant aggregators answer the cost of breadth by moving the hard part *off* the chain entirely — into proprietary solver networks, request-for-quote servers, and keeper bots that index liquidity and push routes on-chain. The chain becomes a settlement venue for decisions made elsewhere, by software the user cannot see and cannot verify.

BlazePhoenix rejects both halves of that bargain. It refuses the linear integration surface by proving that the surface is not linear at all, and it refuses the off-chain escape hatch by keeping the entire decision — discovery, quoting, scoring, routing and settlement — inside contracts that anyone can read and that produce the same answer on every chain. What follows is the design of a protocol whose economic logic is not merely deployed on a blockchain but is *native* to one.

1.1 The fragmentation problem

The fragmentation is real, but it is shallower than it appears. Beneath the surface diversity, the automated market makers that hold the overwhelming majority of on-chain liquidity are governed by a small set of invariants. Constant-product pools obey $x \cdot y = k$. Concentrated-liquidity pools track a square-root price in fixed-point arithmetic and a piecewise-constant active liquidity. Stable-swap pools solve a Newton iteration over an amplified invariant. The Solidly family obeys a quartic, $k = x^3y + xy^3$. There are perhaps six distinct shapes — not sixty.

If the shapes are few, then the cost of aggregation need not grow with the number of venues. It need only grow with the number of *shapes*. A protocol that implements each invariant once, correctly, and dispatches every venue to the right one collapses the integration surface from linear to constant. Adding a venue stops being an act of engineering and becomes an act of configuration: choose a shape, choose an address-derivation mode, and the existing math prices it, the existing router settles it, the existing registry learns it.

CENTRAL CLAIM

Every venue BlazePhoenix supports is a parametrisation of one of a small family of closed-form output functions. The protocol stores those functions in a single library and never redefines them. Adding a venue means choosing a shape and an address-derivation mode — not writing new math. **Six shapes, not sixty.**

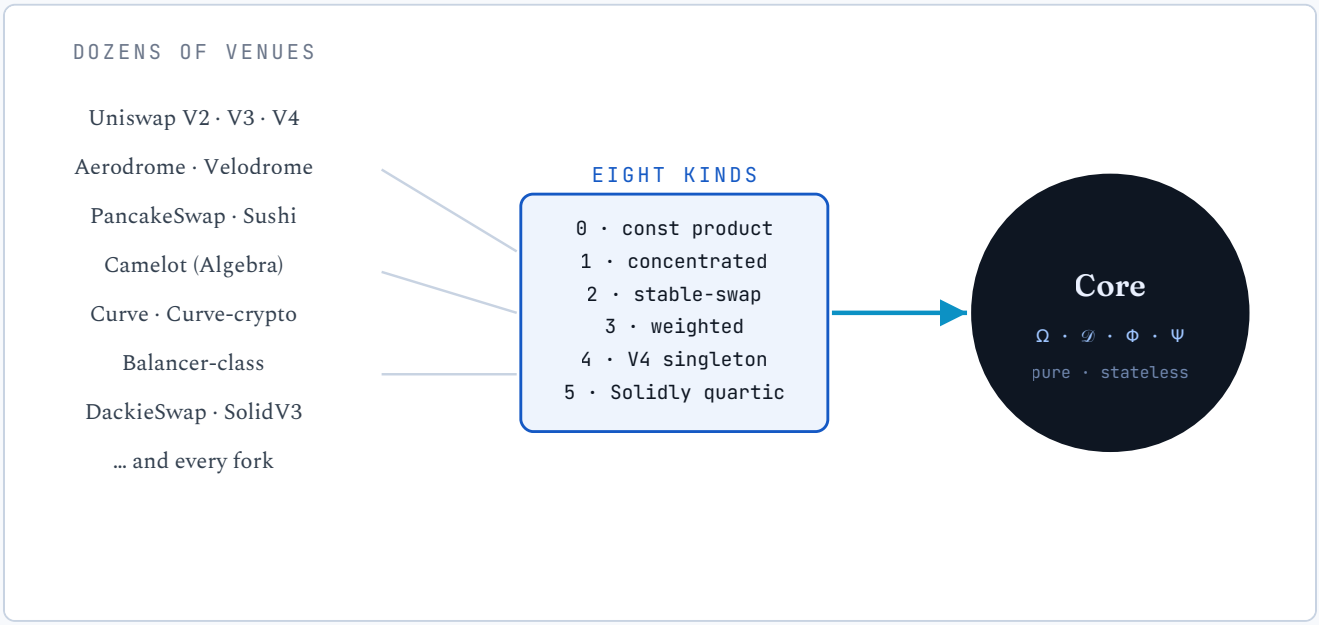

FIG. 1 *The collapse. Dozens of venues map onto a handful of closed-form shapes, which a single stateless library — the Core — prices, locates, scores and floors. Breadth becomes a configuration problem, not an engineering one.*

1.2 Web 2.5 crypto vs Web 3.0

It is worth naming the thing BlazePhoenix is built against, because most of what calls itself decentralized finance is, in its load-bearing parts, neither. The prevailing aggregator architecture quotes routes on a fleet of servers, runs an auction among off-chain solvers, indexes pool liquidity with keeper bots, and ships the result to a thin on-chain settlement contract — frequently behind an upgradeable proxy whose admin key can replace the logic outright. The marketing says Web 3.0; the topology says Web 2.5. A web service produces the price, a database holds the liquidity graph, a privileged operator can change the rules, and the chain is reduced to a payment rail.

That arrangement is not merely inelegant; it reintroduces exactly the trust the technology was meant to remove. An off-chain quote can differ from the on-chain fill, and the user has no way to prove it should not have. A keeper can go dark and the router goes blind. A proxy admin can, in a single transaction, turn an honest contract into a hostile one. None of these are hypothetical failure modes — each has drained real value from real users.

TABLE 1 — TWO ARCHITECTURES FOR THE SAME PROBLEM

Concern	Web 2.5 aggregator	BlazePhoenix
Quoting	Off-chain servers / solver auction	On-chain, same function as execution
Liquidity discovery	Keeper bots index & push lists	Self-healing registry; learned by trading
Route selection	Proprietary off-chain optimiser	Pure on-chain Solver, fully reproducible
Upgrade surface	Admin-controlled proxy	Stateless Core; no upgrade path
Privileged power over funds	Often present	None — Router holds nothing at rest
Cross-chain consistency	Per-chain backend logic	Identical bytecode every chain
Trust assumption	The operator's honesty	The user's signed minimum-output

BlazePhoenix takes the harder road precisely where the easy road hides the risk. The cost is real — quoting a concentrated pool on-chain is more expensive than reading it from a database, and proving an overflow bound is more work than assuming it away — but the cost buys the elimination of an entire class of failure: the silent, unprovable divergence between what a server promised and what the chain delivered. Section 16 returns to this thesis in full; we raise it here because every design decision in the sections between is an instance of it.

1.3 Design philosophy

BlazePhoenix takes four positions, each inherited from the lineage of protocols that prized legibility over feature-count — the spare determinism of Bitcoin's original design and the invariant-first clarity of the Uniswap papers.

PRINCIPLE I

One source of truth

All pricing, address and scoring mathematics lives in a single contract, the Core. The Hub, Solver, Router and Quoter import it; none redefines a primitive. A change happens in exactly one place and every consumer inherits it atomically.

PRINCIPLE II

Quote equals execution

The number shown off-chain and the number computed on-chain are produced by the same function over the same live state. Where a venue charges a dynamic fee, both paths read it from the same source at the same block. There is no separate quoting model to drift out of sync with reality.

PRINCIPLE III

Compute, don't trust

Where an address can be derived, it is derived. Where a bound can be proven, it is proven. The protocol calls external contracts only for the irreducible state — reserves, prices, liquidity — that cannot be known in advance, and verifies even those against what it can compute.

PRINCIPLE IV

No privileged path to funds

The Solver and Quoter are pure read-path functions. The Router is the only contract that moves money, and it moves it atomically under the user's own bound. Every power that could redirect or freeze the protocol can be permanently surrendered.

Throughout, a single discipline recurs: *compute, do not trust*. Where an address can be derived, it is derived; where a bound can be proven, it is proven; where a number can be produced by the same function on both sides of the on-chain/off-chain boundary, it is. The remainder of this paper is that discipline, applied.

1.4 Contributions

This paper documents a working protocol, validated on Ethereum and three Layer-2 networks from a single identical bytecode. Its contributions, each developed in a dedicated section, are summarised below.

TABLE 2 — SUMMARY OF CONTRIBUTIONS

§	Contribution	What is new
3–8	Four universal operators	One closed-form per AMM shape (Ω), one CREATE2 polynomial ($\mathscr{D}$), one pure feasibility floor (Φ), one log-linear vitality field (Ψ) — composed into a single meta-equation.
5	Convergent Solidly solver	Newton seeded at the opposite reserve with a bidirectional half-step clamp; matches live `getAmountOut` within 0.5% where the naïve constant-sum seed silently saturates.
9	The Monoslot	A pool's entire state — vitality, depth bucket, fee, kind, two volume EMAs, four block/time fields — packed into one 256-bit word, read in a single `SLOAD`.
10	Self-healing registry	Liquidity learned by executing it; no keeper, no indexer, no manual lists. Vitality-ranked eviction with a strict 25% admission margin keeps the healthiest pools per pair.
11	Capital-anchored split	A marginal-rate filter anchored on the deepest pool by real output-token balance separates price quality from depth, defeating dust-pool median manipulation.
12	One-callback execution & the Exact Pass	One bounded swap-callback fallback handles every V3-shaped venue; one unlock-settle dance handles V4 singletons; concentrated quotes are taken from the pool by revert-extraction.
15–16	The One-Way Door	A two-tier renunciation that can permanently burn every fund-touching and freeze power while keeping the grow-only registry alive — autonomy as a contract property, not a promise.

2 Invariant-Driven Design

Before the mathematics, a word on method — because the way BlazePhoenix is built is itself a claim about how on-chain systems ought to be built. We call the approach *invariant-driven design*: a discipline in which the protocol is specified not as a sequence of operations but as a set of properties that must hold, with the code organised so that those properties are provable, testable, and inherited everywhere from a single source.

Most software is written imperatively. It describes what to do, step by step, and correctness is an emergent hope: if every step is right and they compose, the whole behaves. For ordinary software this is acceptable, because bugs are recoverable — a patch ships, a server restarts, a database is rolled back. On a public blockchain, none of that holds. Code is immutable once deployed; it custodies value directly; it executes in an adversarial environment where every edge case is actively hunted; and a single arithmetic slip is not a crash but an irreversible loss. The imperative habit — trust the steps, hope they compose — is precisely the wrong habit for this medium.

2.1 What an invariant-driven system is

An invariant is a property that must be true regardless of how the system arrived at its current state. “The Router holds no funds between transactions.” “A quoted output equals the realised output.” “No reserve product can overflow 256 bits.” “A registry holds at most sixteen pools per pair, and they are the healthiest sixteen.” Each is a statement about every reachable state, not about a particular execution path.

An invariant-driven system is one designed so that these statements are *structural* — enforced by the shape of the code rather than by the vigilance of the programmer. Three principles characterise the approach, and all three run through every section of this paper.

TABLE 3 — THE THREE PRINCIPLES OF INVARIANT-DRIVEN DESIGN

Principle	Statement	Where it appears
Compute, don't trust	Where a value can be derived or proven, never fetch or assume it.	CREATE2 derivation (§6); proven overflow bounds (§14); on-chain fee/floor re-derivation (§13).
One source of truth	Every property is defined once and inherited; never restated, never able to drift.	The Core library, imported by all four contracts (§18).
Fail closed	When a property cannot be guaranteed, halt rather than proceed on a guess.	Solver returns an empty route on unquotable pools; the stable solver returns zero rather than a saturated quote; the coherence guard reverts impossible venues (§10.4).

2.2 Why this is the right discipline for crypto

The case is not aesthetic. Each property of the blockchain environment that makes imperative code dangerous is precisely a property that invariant-driven code turns to advantage.

Immutability rewards provability.

Because deployed code cannot be patched, the cost of a latent bug is unbounded and the value of an up-front proof is correspondingly high. Imperative code amortises correctness over many future fixes; on-chain, there are no future fixes. A system whose invariants are established before deployment — the overflow bounds of §14, the quote-execution identity of §12 — converts the immutability that punishes imperative code into a guarantee that, once proven, holds forever.

Composability rewards a single source of truth.

On-chain protocols are called by other protocols, in combinations their authors never anticipated. An aggregator's pricing function may be invoked inside a liquidation inside a vault inside a structured product. If pricing is restated in three places, those three will eventually disagree, and the disagreement will surface as a loss in some composition no one tested. By defining every primitive once in the Core and importing it everywhere, BlazePhoenix makes the disagreement impossible: there is one pricing function, and every caller — internal or external — gets the same answer. Composability stops being a risk and becomes a multiplier.

Adversariality rewards failing closed.

Every input to an on-chain function is potentially hostile, and an attacker needs only one path the author did not consider. An imperative system that proceeds on a best guess when its assumptions break hands that path to the attacker. An invariant-driven system that fails closed — the Solver returning an empty route rather than a fabricated one, the stable solver returning zero rather than a saturated quote, the configuration guard reverting a structurally impossible venue before any state is written — denies the attacker the edge case, because the edge case resolves to a safe halt rather than an exploitable improvisation.

Value-at-stake rewards legibility.

When code custodies money, it must be auditable by parties who did not write it. A system expressed as a small set of named properties over a single mathematical core can be read, checked, and reasoned about in a way that a sprawl of special-cased operations cannot. The four operators of §3–§8 are not merely an implementation; they are the protocol's specification, written in a form an auditor can verify against the code line by line.

> THE ARGUMENT IN ONE LINE
>
> Imperative code asks “did I handle every case?” — a question no author can answer with certainty in an adversarial, immutable, composable environment. Invariant-driven code asks “is this property true in every state?” — a question that can be proven once and trusted forever. For software that cannot be patched and holds value under attack, the second question is the only safe one to build on.

2.3 The cost, honestly stated

The discipline is not free, and it would be dishonest to present it as such. Establishing an invariant up front is more work than writing the imperative path that usually suffices. Proving an overflow bound takes longer than assuming it will not happen. Routing every product through a full-precision multiply-divide primitive costs gas that a naïve multiply would not. Insisting that a pool address be derived and verified rather than simply fetched adds engineering that a less careful protocol skips.

What the discipline buys in return is the elimination of an entire class of failure — the silent, irreversible, composition-triggered loss that has drained more value from this industry than any other single cause. The trade is deliberate: more cost before deployment, in exchange for the removal of the failures that cannot be recovered after it. For a protocol that intends to custody

value on an immutable ledger, we hold that this is not merely a reasonable trade but the only responsible one. The remainder of this paper is the discipline applied: every operator a proven property, every property defined once, every uncertainty resolved by halting rather than guessing.

3 The Meta-Equation

The whole protocol can be read as the maximisation of a single expression. Given an input token t_{in}, an output token t_{out} and an amount x, the Solver searches the space of admissible routes for the one that maximises net received value subject to a feasibility floor and an anti-scam gate:

META-EQUATION — THE OPTIMISATION TARGET

$$\mathcal{U}(\text{route}) \;=\; \mathbb{1}\Big[\, y_{\text{net}} \geq \Phi(\iota, \ell, \sigma) \cdot y_{\text{quote}} \,\Big] \cdot \Xi(\text{route}) \cdot y_{\text{net}} \cdot \prod_{i} \Psi_i^{\,w_i} \tag{1}$$

Each symbol is a universal operator, defined exactly in the sections that follow. The net output $y_{\text{net}} = \big(\sum_i \Omega(\text{pool}_i, x_i)\big) - \text{fee}$ is the sum of the per-leg outputs produced by the dispatcher, less the protocol fee; ι is the route's measured price impact, ℓ its leg count, and σ a log-variance hint.

TABLE 4 — THE FOUR UNIVERSAL OPERATORS

Operator	Name	Signature	Role in (1)
Ω	Eightfold dispatcher	$(\text{pool}, x) \mapsto (y, \mathcal{L}, \iota)$	Computes each leg's output y
$\mathscr{D}$	Deterministic derivation	$(\text{factory}, t_0, t_1, \ldots) \mapsto \text{pool}$	Locates the pools the route uses
Φ	Iron-Law floor	$(\iota, \ell, \sigma) \mapsto \text{bps}$	Projects the route onto the feasible set
Ψ	Vitality field	$(\text{slot}) \mapsto \text{score}$	Weights each pool by quality

The factorisation is deliberate. **Ω** answers how much a pool returns; **Ψ** answers how much to trust it; **Φ** answers whether the route is admissible at all; $\mathscr{D}$ answers where the pool lives; and **Ξ** — the anti-scam projector of §15 — answers whether the route is safe to surface. The product form means a single failing leg $(\Psi_i = 0)$ collapses the route's score, while the indicator $\mathbb{1}[\cdot]$ hard-zeroes any route whose net output falls below the dynamic floor — no soft penalties, no tunable knobs, no place for a near-miss route to sneak through.

WHY A SINGLE EQUATION

An aggregator's value is one number — the output the user receives — disciplined by two binary questions: is the route feasible, and is it safe? Expressing the whole engine as one expression makes those questions impossible to forget. There is no separate scoring heuristic to drift, no penalty weight to mistune; a route either clears the floor and the gate or it scores zero.

4 The Eightfold Dispatcher Ω

The dispatcher is the heart of the protocol. It takes a pool of any kind and an input amount and returns three quantities: the output y, a depth proxy $\mathcal{L}$ used by the split allocator, and a price-impact hint ι. Internally it branches on the pool's kind — an eight-value enumeration — and applies the corresponding closed-form function or, where a venue's own bytecode is the only honest source, asks the pool directly. We give each branch below, exactly as the Core computes it.

4.1 Constant product (Uniswap V2 & forks)

The simplest invariant. Reserves satisfy $r_{\text{in}} \cdot r_{\text{out}} = k$, and a fee f (in basis points, $10^4 = 100\%$) is taken from the input. The exact output is

CONSTANT-PRODUCT OUTPUT

$$y = \frac{x\,(10^4 - f)\,r_{\text{out}}}{10^4\,r_{\text{in}} + x\,(10^4 - f)} \tag{2}$$

This is computed in a single expression with no division until the final step, so the only rounding is the floor of the last quotient. The depth proxy is $\mathcal{L} = \min(r_{\text{in}}, r_{\text{out}})$. When a venue exposes no fee in its registration (the empty-fee case), the dispatcher defaults to 30 bps — the universal Uniswap-V2/Sushi fee — so that the quoted output matches the pool's own $x \cdot y = k$ check and the swap cannot revert on a fee mismatch. The same default is mirrored byte-for-byte in the execution path, so quote and fill agree by construction.

4.2 Concentrated liquidity (Uniswap V3, Algebra)

Concentrated-liquidity pools do not store reserves; they store a square-root price $\sqrt{P}$ in Q64.96 fixed point and an active liquidity L. Within a tick the swap follows the canonical sqrt-price update. For token0 → token1 (`zeroForOne`):

V3 SQRT-PRICE UPDATE & OUTPUT, TOKEN0 → TOKEN1

$$\sqrt{P}' = \frac{L\sqrt{P}}{L + \dfrac{x(1-f)\sqrt{P}}{Q_{96}}}, \qquad y = \frac{L\left(\sqrt{P} - \sqrt{P}'\right)}{Q_{96}} \tag{3}$$

where f here is the V3 fee in parts-per-million and $Q_{96} = 2^{96}$. The reverse direction (token1 → token0) uses the dual update $\sqrt{P}' = \sqrt{P} + x(1-f)\,Q_{96}/L$ followed by $y = L\,Q_{96}\,(\sqrt{P}' - \sqrt{P})/(\sqrt{P}\,\sqrt{P}')$. The naïve computation overflows: the product $L \cdot \sqrt{P}$ can reach 2^{288}, thirty-two bits beyond the 256-bit word. The Core never forms it — every product is factored through a full-precision `mulDiv` primitive (a 512-bit intermediate, 256-bit result), so each stage provably fits. This is treated in full in §14.

Equation (3) is exact within a single tick. It holds to the unit on deep pools, where a trade does not exhaust the active liquidity, and the dispatcher uses it directly for scoring and split allocation — the regime that covers the overwhelming majority of routed volume. It is not exact across tick boundaries: in a concentrated pool a sufficiently large trade crosses initialised ticks where L changes, and a single-tick formula then overestimates the realisable output. **BlazePhoenix deliberately does not attempt to replicate the pool's tick-crossing arithmetic.** Reproducing a venue's internal mathematics is a structural class of bug: any divergence between the copy and the original surfaces as a mispriced quote and a rejected or adverse fill. Instead the binding quote for a concentrated leg is taken from the pool itself, by the revert-extraction mechanism of §12.4. The closed form prices and ranks; the pool, asked directly, settles the exact number.

A PRINCIPLE, NOT A SHORTCUT

The decision to price-and-rank with a closed form but settle with the pool's own swap is the single most important correctness choice in the dispatcher. It means the number a trader is shown for a concentrated venue is not an approximation that might drift — it is the venue's own bytecode, run over live state and rolled back. Replication bugs are not mitigated; they are made impossible, because there is no replica.

4.3 Stable-swap and Solidly invariants

Two further shapes serve correlated assets, where the constant-product curve wastes capital by pricing near-equal tokens as if they could diverge arbitrarily. The Curve-style stable-swap blends constant-sum and constant-product through an amplification coefficient A, defining the invariant

CURVE STABLE-SWAP INVARIANT

$$A n^n \sum_i x_i + D = A D n^n + \frac{D^{n+1}}{n^n \prod_i x_i} \tag{4}$$

The Solidly family — Aerodrome, Velodrome and their forks — instead uses a quartic invariant that hugs the line $x = y$ far more tightly than constant product while still admitting large divergence:

SOLIDLY STABLE INVARIANT

$$k = x^3 y + x y^3 = x y (x^2 + y^2) \tag{5}$$

Figure 2 shows why this matters. Near the balance point, the Solidly curve delivers almost the full input as output — minimal slippage — precisely where stablecoin and liquid-staking trades cluster. The constant-product curve, by contrast, begins losing value immediately. The output of a Solidly stable swap is the difference between the pre- and post-trade output reserves, $y_{\text{out}} = r_{\text{out}} - y^\star$, where $y^\star$ solves equation (5) at the post-trade input balance. That solve is the single most delicate computation in the protocol, and §5 is devoted to it.

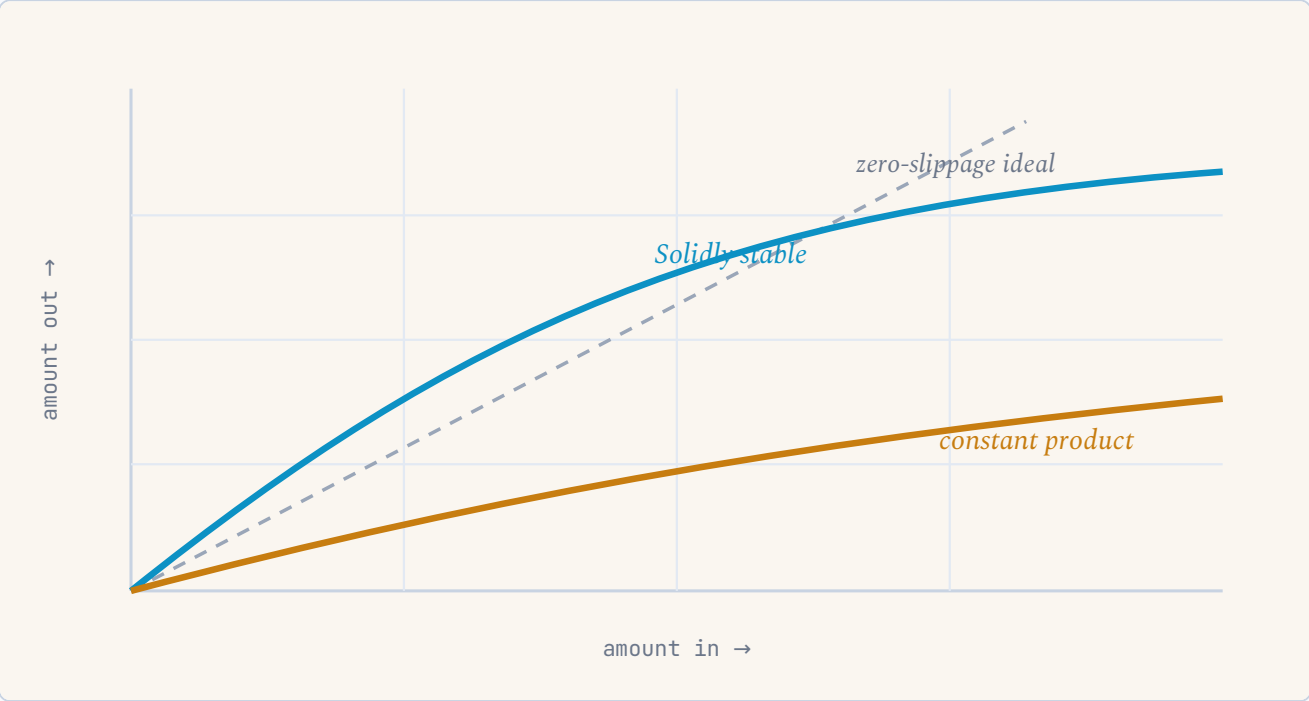

FIG. 2 *Output as a function of input for a balanced pool ($r_{\text{in}} = r_{\text{out}}$). The Solidly stable invariant (teal) tracks the zero-slippage ideal far longer than constant product (ember); near the balance point it returns almost the full input. Solving equation (5) for $y^\star$ is the subject of §5.*

4.4 The Curve adapter — ask, never replicate

Curve's stable-swap and crypto invariants have no closed form simple enough to reproduce safely, and reproducing them is exactly the replication-bug risk the protocol refuses elsewhere. So for Curve venues the dispatcher does not solve equation (4) at all. It *asks the pool*. The Core resolves the coin indices by scanning the pool's `coins()` interface — tolerating both the `uint256` and `int128` signatures that different Curve generations expose — and then calls the pool's own `get_dy(i, j, dx)` for the quote. Because the quote function and the execution function (`exchange`) read the same pool state, they cannot diverge: the number previewed is the number the swap will produce.

This *ask, never replicate* doctrine recurs throughout the protocol — in the Curve adapter, in the concentrated-venue exact pass, and in the fee-on-transfer recompute. Wherever a venue is the only honest authority on its own output, BlazePhoenix treats the venue's bytecode as the specification and reads from it rather than guessing. The cost is a staticcall; the benefit is the structural impossibility of a quote-versus-fill mismatch.

4.5 Uniswap V4 via state override

Uniswap V4 abolishes the per-pool contract: every pool lives inside one `PoolManager` singleton, addressed by a `poolId` that is the hash of its `PoolKey` $(c_0, c_1, \text{fee}, \text{tickSpacing}, \text{hooks})$. There is no pool address to call `slot0()` on. The Core instead reads the manager's storage directly, by `extsload`: the pool's base slot is $\text{keccak256}(\text{abi.encode}(\text{poolId}, 6))$, its packed `slot0` carries the sqrt-price in the low 160 bits, and the active liquidity sits three words later. This slot layout was verified against the canonical V4 `StateView` on a mainnet fork, not assumed from documentation.

With $\sqrt{P}$ and L in hand, the dispatcher prices a V4 leg with the same concentrated-liquidity formula of equation (3) — a current-tick approximation that ignores tick-crossing and ignores the hook, which is acceptable for scoring because the Router rejects delta-altering hooks (§15.1) and the Iron-Law floor catches large-swap divergence. The binding V4 quote, like every concentrated quote, is taken from the pool by revert-extraction (§12.4). To the Solver and to the trader, a V4 leg is indistinguishable from any other.

4.6 The dispatch table

The dispatcher branches on an eight-value `kind` enumeration. Each kind maps to one output function and one depth proxy. This table is the entire venue-integration surface of the protocol: adding a venue means assigning it a kind, not writing pricing code.

TABLE 5 — THE UNIVERSAL DISPATCH: EVERY VENUE, ONE OF EIGHT KINDS

Kind	Family	Output function	Depth proxy $\mathcal{L}$	Launch status
0	Uniswap V2 & forks	Constant product, eq. (2)	$\min(r_{\text{in}}, r_{\text{out}})$	live
1	Uniswap V3	Sqrt-price, eq. (3)	active liquidity L	live
2	Curve stable-swap	On-chain `get_dy`	quoted out	adapter[1]
3	Balancer-class	Constant-product kernel[2]	$\min(r_{\text{in}}, r_{\text{out}})$	kernel[2]
4	Uniswap V4	State-override sqrt-price	active liquidity L	live
5	Solidly / Aerodrome	Quartic invariant, eq. (5)	$\min(r_{\text{in}}, r_{\text{out}})$	live
6	Algebra (Camelot V3)	Sqrt-price, dynamic fee	active liquidity L	live
7	Curve-crypto	On-chain `get_dy`	quoted out	adapter[1]

[1] Curve kinds (2, 7) are library-complete — quoted by `get_dy` and settled by `exchange` via the dual-signature adapter — but are admitted to the registry only through the dedicated Curve meta-pool mode and are excluded from the validated launch venue set pending their own fork-test coverage (§19). [2] Balancer-class (kind 3) is dispatched through the constant-product kernel, exact for balanced weightings; the full weighted-invariant solver is a roadmap extension.

Two distinct fee regimes coexist. Constant-product and Solidly venues quote fees in basis points ($10^4 = 100\%$); concentrated-liquidity venues quote in parts-per-million ($10^6 = 100\%$). The dispatcher carries each in its native unit and never cross-converts, eliminating a class of rounding bug. For Solidly venues the dispatcher additionally reads the *live* fee the pool will charge at execution — which a gauge may set above the configured default — so the quoted fee and the executed fee are by construction identical.

5 The Solidly Solver

Equation (5) is quartic in y and has no useful closed form, so we solve it numerically. Given the post-trade input balance x and the target invariant K computed from the pre-trade reserves, we seek the root of

ROOT-FINDING TARGET

$$f(y) \;=\; k(x,y) - K \;=\; x^3y + xy^3 - K \;=\; 0 \tag{6}$$

Newton's method gives $y_{n+1} = y_n - f(y_n)/f'(y_n)$. The method is only as good as its seed, and here the seed is everything.

5.1 Newton seeding and convergence

A natural-looking seed is $y_0 = K/x$ — the constant-sum approximation. It is also wrong in a way that silently corrupts prices. For a pool absorbing an input several times its reserve, that seed places y_0 far above the true root on a region where f is extremely steep; the iteration overshoots, the integer division floors the step to zero, and the solver *saturates* — returning the full opposite reserve as though the trade drained the pool. The quote is catastrophically wrong, yet no exception is raised. It is the worst kind of bug: a confident, plausible, completely incorrect number.

> THE FIX THAT EMPIRICAL TESTING FORCED
>
> Seeding instead at $y_0 = Y$, the opposite pre-trade reserve, and stepping bidirectionally — up when $k(x,y) < K$, down when $k(x,y) > K$, and never past zero — converges for both balanced and heavily-skewed pools. Against Aerodrome's live `getAmountOut`, this reproduces the on-chain output to within 0.5%, and exactly for balanced pools.

The iteration runs for at most 64 steps with an early exit when successive iterates differ by at most one unit. The derivative $f'(y) = x\,(x^2 + 3y^2)$ is positive for all $x, y > 0$, so each step is well-defined whenever $f' \neq 0$; the solver bails to zero — treating the pool as unquotable — rather than divide by a vanishing derivative. The downward step is additionally clamped to at most half the

current iterate, so y provably never crosses zero. Figure 3 contrasts the two seeds on a real stable pool: the opposite-reserve seed descends monotonically to the true root in a handful of iterations; the constant-sum seed climbs to a fixed point pinned at saturation and never recovers.

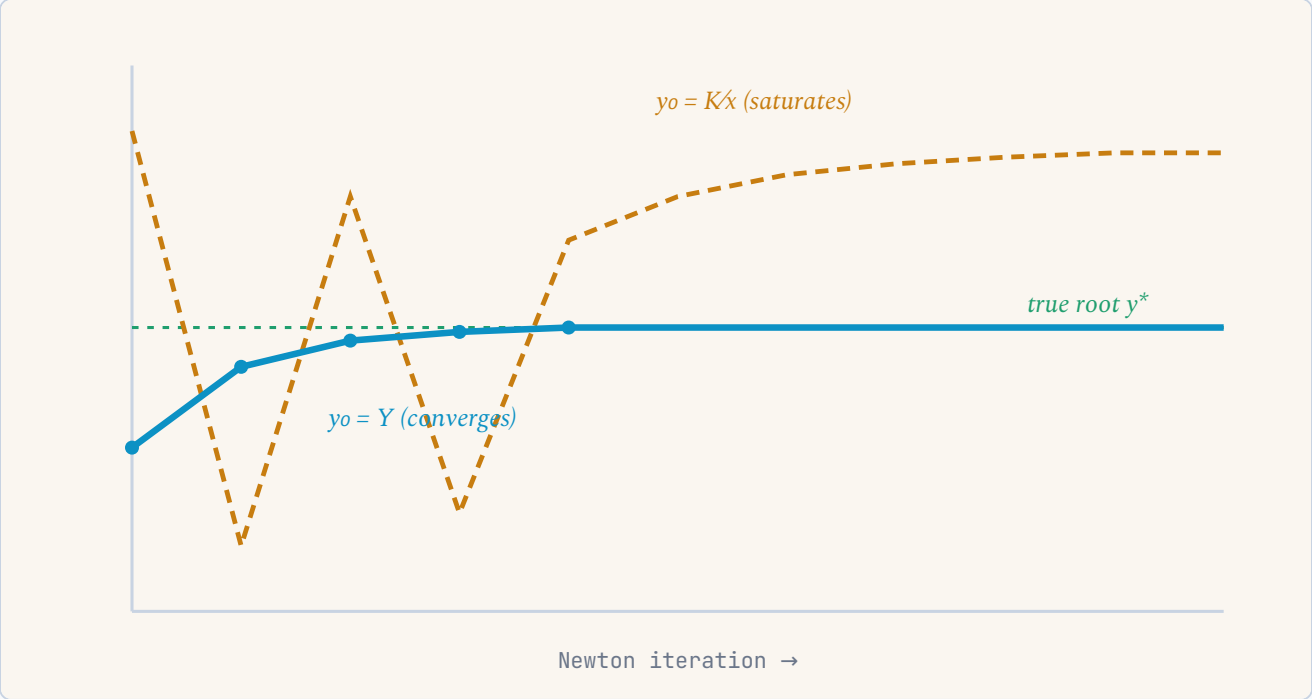

FIG. 3 *Newton iterates for the Solidly root solve on a real stable pool. Seeding at the opposite reserve Y (teal) converges to the true root $y^\star$; the constant-sum seed K/x (ember, dashed) oscillates and pins at saturation, yielding a corrupt quote. The bidirectional step with a half-step clamp guarantees the iterate never crosses zero.*

5.2 Decimal normalisation

The invariant $k = xy(x^2 + y^2)$ is homogeneous of degree four. When both tokens share decimals — USDC and USDbC, both six — the scale cancels and raw reserves can be used directly. But a pair such as DOLA (18 decimals) against USDC (6) carries a 10^{12} scale mismatch into the quartic, and the homogeneity no longer saves us: the cubic terms see wildly different magnitudes and the solve degrades. The Core therefore normalises. Each reserve is scaled to a common 10^{18} basis before the solve, and the output is de-scaled afterwards:

DECIMAL-NORMALISED STABLE SOLVE

$$X = r_{\text{in}} \cdot 10^{18-d_{\text{in}}}, \quad Y = r_{\text{out}} \cdot 10^{18-d_{\text{out}}}, \quad y_{\text{out}} = \frac{Y - y^\star(X + A,\, K)}{10^{18-d_{\text{out}}}} \tag{7}$$

where $A = x \cdot 10^{18-d_{\text{in}}}\,(1 - f/10^4)$ is the normalised post-fee input and $K = k(X, Y)$. The decimals $d_{\text{in}}, d_{\text{out}}$ are read on-chain from each token, defaulting to 18 for tokens that do not expose `decimals()`. The equal-decimal case skips normalisation entirely on a fast path, so the common USDC/USDbC trade pays nothing for a generality it does not need.

5.3 Validation and the under-quote margin

Table 6 reports the Core stable solver against three live Aerodrome pools spanning balanced and skewed states.

TABLE 6 — CORE STABLE SOLVER VS. AERODROME LIVE GETAMOUNTOUT

Pool	State	Ground truth	Core output	Deviation
USDC / USDbC	balanced (6/6 dec)	999 534 071	999 534 071	0.000%
DOLA / USDC	skewed (18/6 dec)	985 528 437	987 998 286	0.251%
USDz / USDC	skewed (18/6 dec)	899 325 141	901 565 830	0.249%

All three fall inside the 0.5% tolerance, and the balanced pair matches to the integer. The protocol additionally under-quotes Solidly venues by a fixed 200-basis-point margin — returning $0.98\,y_{\text{out}}$ — to absorb the rounding semantics of Aerodrome's on-chain k-invariant check, which uses post-fee balance reads whose rounding the canonical formula does not capture exactly. Under-quoting guarantees the pool always has slack to satisfy its invariant at execution time: it sacrifices a small slice of headline depth but eliminates the over-quote race that would otherwise revert the swap. Because the same 200-bps margin is applied identically in the quote and execution paths, the previewed number and the realised number remain in lock-step. Crucially, this margin is paid back to the trader: any output realised above the (deliberately conservative) quote is surplus, and §13 returns 100% of it to the user, fee-exempt.

6 Deterministic Derivation $\mathscr{D}$

Before a pool can be quoted, it must be located. Most aggregators ask the factory: a call to `getPool(t0, t1, fee)` reads a storage mapping and returns the address. BlazePhoenix supports this — four factory-call modes cover the venues whose deployment is not reproducible — but prefers, where possible, to compute the address with no external call at all. An address that is computed is an address that can be *verified*; an address that is merely fetched can only be trusted.

6.1 The CREATE2 polynomial

The EVM's `CREATE2` opcode makes a contract's address a pure function of three inputs: the deployer, a salt, and the hash of the contract's init code. If all three are known, the address is reproducible by arithmetic alone:

CREATE2 ADDRESS DERIVATION

$$\text{pool} \;=\; \text{last}_{20}\Big(\,\text{keccak256}\big(\,\texttt{0xff} \,\|\, \text{deployer} \,\|\, \text{salt} \,\|\, \text{initCodeHash}\,\big)\Big) \tag{8}$$

The salt encodes the pool's identity. BlazePhoenix unifies the venue shapes by selecting the salt polynomial and the CREATE2 origin from the pool kind, so a single function derives addresses for every reproducible family.

6.2 Salt construction by venue shape

TABLE 7 — DERIVATION MODES: SALT CONSTRUCTION BY VENUE SHAPE

Mode	Family	Salt	Resolution
0	V2	`—`	factory `getPair(t0,t1)`
1	V3	`—`	factory `getPool(t0,t1,fee)`
2	Solidly	`—`	factory `getPool(t0,t1,stable)`
3	V3-CL	`—`	factory `getPool(t0,t1,spacing)`
4	V2 CREATE2	`keccak(t0‖t1)`	CREATE2, origin = factory
5	V3 CREATE2	`keccak(enc(t0,t1,fee))`	CREATE2, origin = factory
5*	Algebra	`keccak(enc(t0,t1))`	CREATE2, origin = `poolDeployer()`
6	EIP-1167 clone	`keccak(t0‖t1‖stable)`	CREATE2, internal deployer
7	V3-CL CREATE2	`keccak(enc(t0,t1,spacing))`	CREATE2, origin = factory

Two subtleties earned their place through empirical reverse-engineering. The Algebra family (Camelot V3) charges a *dynamic* fee, so its salt omits the fee entirely — using `keccak256(abi.encode(t0,t1))` — and its pools are deployed not by the factory but by a separate `poolDeployer` contract, which the Core resolves via a single `staticcall` (selector `0x3119049a`), falling back to the factory if the selector is absent. A plain V3 factory in the same mode therefore still derives correctly. A second fallback covers Algebra factories that expose `poolByPair(t0,t1)` rather than the canonical `getPool`. After every derivation, the Hub applies a `hasCode` guard: any derived address with no bytecode is discarded, so a wrong hash can never silently locate a non-existent pool.

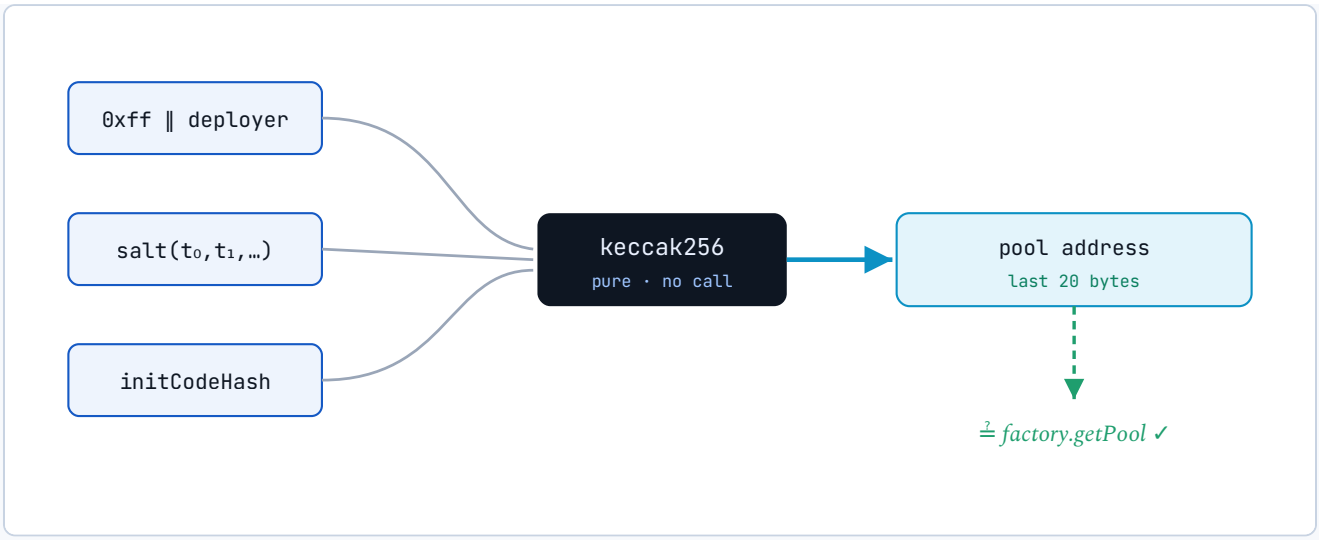

FIG. 4 *Deterministic derivation. Three known inputs hash to the pool's address with no external call. The result is then checked against the factory's live* `getPool` *— a standing invariant of the test suite — so a derived address is a statement the protocol can verify, not a guess it must trust.*

6.3 Verified derivation across chains

The protocol's flagship derivation is Uniswap V3. Its init-code hash is a single constant — identical on every chain, because the same pool bytecode is deployed everywhere — which makes V3 the one venue family the protocol derives by pure arithmetic in production, on every network, with no factory call on the discovery path. The claim is not assumed; it is checked. For every fee tier and every chain, the derived address of equation (8) is compared against the factory's live `getPool`, and the equality is a standing invariant of the test suite.

TABLE 8 — CREATE2 DERIVATION VERIFIED AGAINST LIVE GETPOOL

Chain	Pair / tier	Derived = on-chain?
Ethereum	WETH/USDC, all tiers	match
Base	WETH/USDC, all tiers	match
Arbitrum	WETH/USDC 0.05% → 0xC6962…	match
Optimism	WETH/USDC 0.05% → 0x1fb3c…	match

On all four chains, Uniswap V3 is registered in CREATE2 mode and routes end-to-end — confirming the derived addresses locate real, swappable liquidity, not merely addresses that happen to match. The clone-based venues (Velodrome, Aerodrome) remain on factory-call: their init-code hash is correct and verifiable, but the CREATE2 origin lives inside an internal

`cloneDeterministic` deployer rather than the factory, so their addresses are not reproducible from the factory alone. The protocol derives where the arithmetic is sound and asks where it is not — a boundary established empirically, pool by pool, and never guessed.

7 The Iron Law Φ

Not every route that can execute should. A route that returns 60% of fair value is worse than no trade at all, and a naïve maximiser — chasing the highest nominal output — will happily select a path through a near-empty pool if the arithmetic momentarily favours it. The Iron-Law floor is the protocol's feasibility projection: a pure indicator that zeroes any route whose net receive falls below a dynamic minimum.

7.1 The retention floor

The floor is expressed as a retention fraction in basis points. It starts tight — a base of 96% for a clean, single-leg, low-impact swap — and loosens with two adversaries of execution quality: price impact and per-leg risk. It never falls below a hard 75% floor; the protocol will never knowingly admit a route losing more than 25% to slippage and fees. The separation between the 96% base and the 75% hard floor is deliberate: it makes the floor genuinely adaptive, so a clean swap is held to a far stricter standard than a deep multi-leg one, while the 25% ceiling remains an inviolable backstop.

IRON-LAW RETENTION FLOOR (BASIS POINTS)

$$\Phi = \max\Big(9600 - 200\,(\ell - 1) - \iota - \tfrac{\sigma}{10^{14}},\ 7500\Big) \tag{9}$$

where ℓ is the leg count, ι the V2-style price impact in basis points, and σ a log-variance hint. The impact itself is computed round-up to stay conservative:

PRICE IMPACT, ROUND-UP

$$\iota = \left\lceil \frac{x \cdot 10^4}{r_{\text{in}} + x} \right\rceil \quad (\text{capped at } 10^4) \tag{10}$$

The function is entirely pure — no storage reads, no oracle, no external call — so it adds no gas-metered state access to the hot path. A route whose net output clears the floor passes through unchanged; one that does not is mapped to zero and drops out of the $\max$ in equation (1).

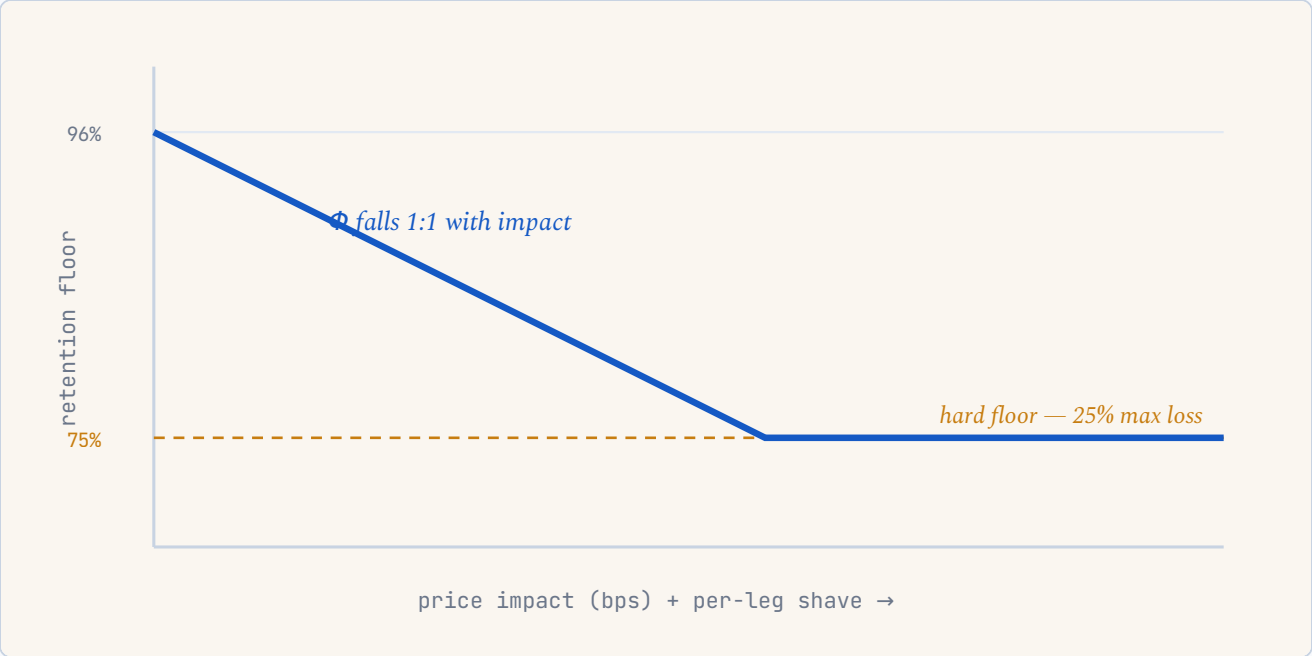

FIG. 5 *The adaptive retention floor. A clean single-leg swap is held to 96%; impact, leg count and volatility loosen the floor linearly, but it is clamped at 75% — the protocol will never knowingly admit a route losing more than a quarter of fair value.*

7.2 On-chain re-derivation

The floor is not merely a quote-time heuristic; it is enforced at execution, and it is enforced on numbers the Router computes itself. The route handed to the swap entry-point carries advisory accounting fields — a quoted total and a suggested floor — but the Router trusts neither for the binding floor. It measures each leg's real price impact from live reserves as the swap executes, recomputes Φ from that measured impact and the true leg count, and applies the floor to the *realised* output. The effective minimum enforced is the strictest of three quantities: the user's own minimum-output bound, any floor the route requested, and this re-derived protocol floor. Caller-supplied fields can only *tighten* protection, never relax it. A route that arrives with a flattering floor cannot use it to slip a bad fill past the user, because the floor the Router enforces is one it derived from the trade that actually happened.

8 The Vitality Field Ψ

The dispatcher tells us what a pool returns now; the vitality field tells us how much to trust that figure when allocating across several pools. A pool may quote well on a tiny probe yet be too shallow to honour a large fill; another may look mediocre at full size yet be perfectly healthy, merely small. Ψ separates quality from capacity, and it does so reading only the pool's own packed state — no external call, no oracle.

8.1 The log-linear field

The Core encodes vitality as a scalar field over a usage-derived activity score v, the pool's depth bucket b, a bridge-token bit, and a concentration flag. These combine multiplicatively — a log-linear score:

VITALITY FIELD

$$\Psi \;=\; v \cdot 2^{\,b} \cdot \left(1 + \tfrac{2500}{10^4}\,[\text{bridge}]\right) \cdot \left(1 + \tfrac{500}{10^4}\,[\text{concentrated}]\right) \tag{11}$$

A pool on a bridge pair earns a 25% bonus — bridges are the connective tissue of the liquidity graph and deserve a routing preference — and a concentrated pool earns a further 5%, reflecting its capital efficiency. The weights w_i in the meta-equation (1) are the normalised vitalities, so a route's score concentrates on its healthiest legs and a single dead leg $(\Psi = 0)$ collapses the product.

8.2 Logarithmic depth buckets

The depth term deserves attention. Rather than carry raw reserves — which would demand a fresh external read on every score — the Core buckets depth logarithmically and stores only the bucket exponent b, recovering a weight by a single shift:

LOGARITHMIC DEPTH BUCKET

$$b \;=\; \min\!\Big(15,\; \big\lfloor \log_{10}(\text{depth}/10^{15}) \big\rfloor\Big), \qquad \text{weight} \;=\; 2^{\,b} \tag{12}$$

so each decade of depth raises the bucket by one and doubles the weight. Depth thus enters the score *sub-linearly*: a deep pool is firmly favoured over a shallow one, but a merely-healthy pool is not erased by a giant. The bucket is four bits, capped at fifteen, and lives in the same packed word as everything else the scorer needs.

8.3 Temporal decay

A pool that was deep and active an hour ago but has since gone quiet should not retain its rank. Vitality therefore decays with idleness. The activity score is the pool's swap count, halved for every sixteen blocks of idle time beyond a sixteen-block grace window:

AGE-DECAYED VITALITY

$$v(\text{age}) \;=\; \begin{cases} \text{swapCount}, & \text{age} < 16 \\ \left\lfloor \text{swapCount} \cdot 2^{-\lfloor \text{age}/16 \rfloor} \right\rfloor, & 16 \le \text{age} < 512 \\ 0, & \text{age} \ge 512 \end{cases} \tag{13}$$

The decay is deliberately responsive: a once-busy pool that goes silent halves in rank every sixteen blocks and falls to zero after 512 idle blocks, ceding its registry slot to fresher liquidity. A single swap revives it instantly — the score resets to the live swap count and the clock restarts. This is the mechanism by which the registry stays current without any keeper, cron, or off-chain process: activity is the only thing that keeps a pool ranked, and activity is observed directly, on-chain, as it happens.

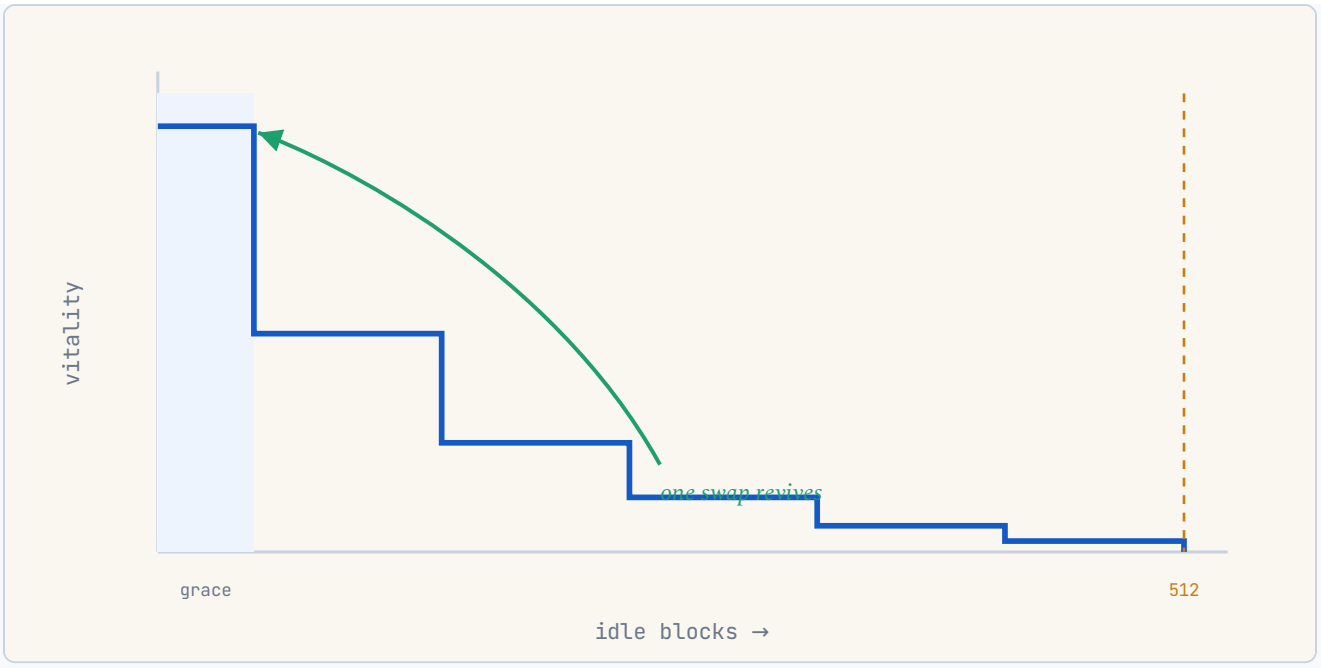

FIG. 6 *Age-decayed vitality. Full rank through a sixteen-block grace, then a halving every sixteen idle blocks, reaching zero at 512 — at which point the pool is fully evictable. A single swap resets the score to the live count and restarts the clock.*

9 The Monoslot

Storage is the dominant cost on the EVM: a cold `SLOAD` is 2,100 gas, and a naïve registry that scattered a pool's attributes across separate slots would pay that cost many times over on every score. BlazePhoenix instead packs a pool's *entire* state into a single 256-bit word — the **Monoslot** — read once, decoded by pure arithmetic.

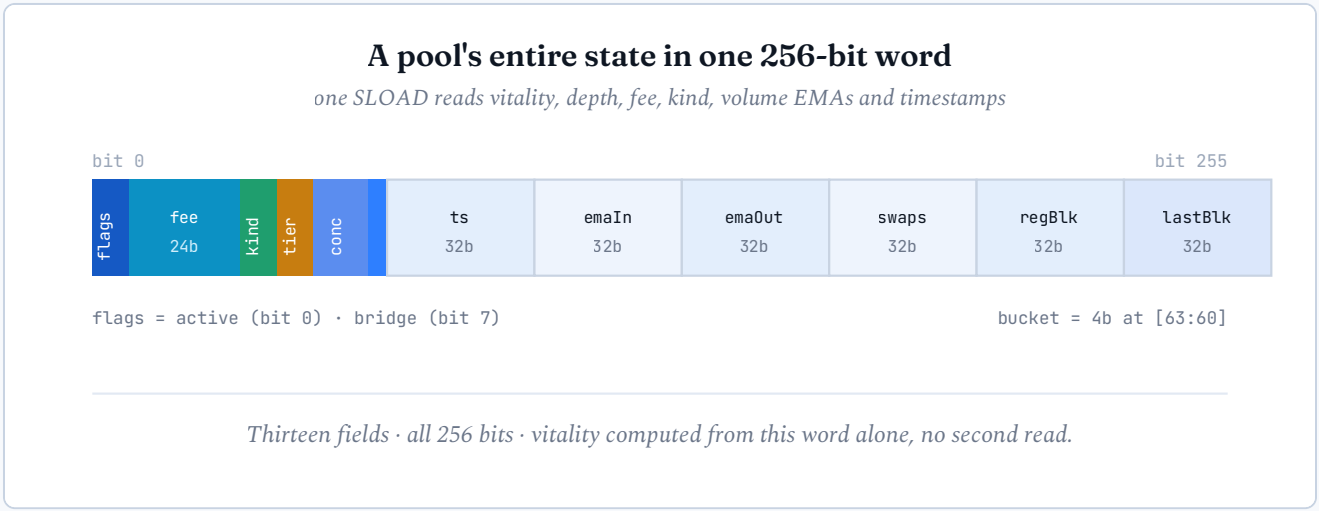

FIG. 7 *The Monoslot. The active and bridge flags, fee, kind, risk tier, concentration bonus, depth bucket, a wall-clock timestamp, two volume EMAs, the swap count, and the registration and last-touch block numbers occupy one 256-bit word, read in a single* `SLOAD`*. Vitality is computed purely from this word; no second read is required.*

The encoding is a sequence of shifts, packing each field at a fixed bit offset:

SLOT ENCODING (BIT OFFSETS)

$$s = \mathtt{active} \,|\, (\mathtt{fee} \ll 8) \,|\, (\mathtt{kind} \ll 32) \,|\, (\mathtt{tier} \ll 40) \,|\, (\mathtt{conc} \ll 48) \,|\, (\mathtt{bucket} \ll 60) \,|\, \cdots^{(14)} \,|\, ($$

The concentration field is masked to twelve bits before shifting so it can never bleed into the depth bucket at bits [63:60] — a deliberate guard against the kind of silent field overlap that corrupts packed state. The two volume fields are exponential moving averages of input and output flow, updated on each successful swap, giving the protocol a decaying memory of a pool's throughput without storing a history. The two block fields — registration and last-touch — drive the temporal decay of equation (13), while the wall-clock timestamp drives the Solver's discovery-freshness gate (§10.3). Because every field the scorer needs lives in this one word, computing a pool's vitality is a single read followed by pure arithmetic: no second slot, no external call, no oracle.

WHY THIS MATTERS

The split allocator scores every candidate pool for every leg of every route it evaluates. Folding the full pool state into one word turns what would be a multi-read, multi-slot scoring loop into a tight arithmetic kernel over single `SLOAD`s — the difference between a routing engine that is affordable to run on-chain and one that is not. The Monoslot is what makes on-chain routing economically possible at all.

10 The Self-Healing Registry

Most aggregators maintain their venue knowledge off-chain: a keeper indexes pools, ranks them, and pushes lists on-chain. That introduces a trusted process, a staleness window, and an operational dependency — three things a protocol that aspires to be native to the chain cannot accept. BlazePhoenix takes the opposite stance: **the registry learns liquidity by executing it.**

10.1 Liquidity learned by trading

On every successful swap leg, the Router calls back into the Hub's `recordSwap`. If the pool is already known, its Monoslot is ticked: the swap count rises, the volume EMAs update, the last-touch block advances, the depth bucket is recomputed, and its vitality is refreshed. If the pool is unknown — and the pair has room, or the newcomer is healthier than the weakest incumbent — it is inserted, with no governance action and no manual list. A pool becomes part of the protocol's knowledge by the only act that proves it exists and works: a swap that settled through it.

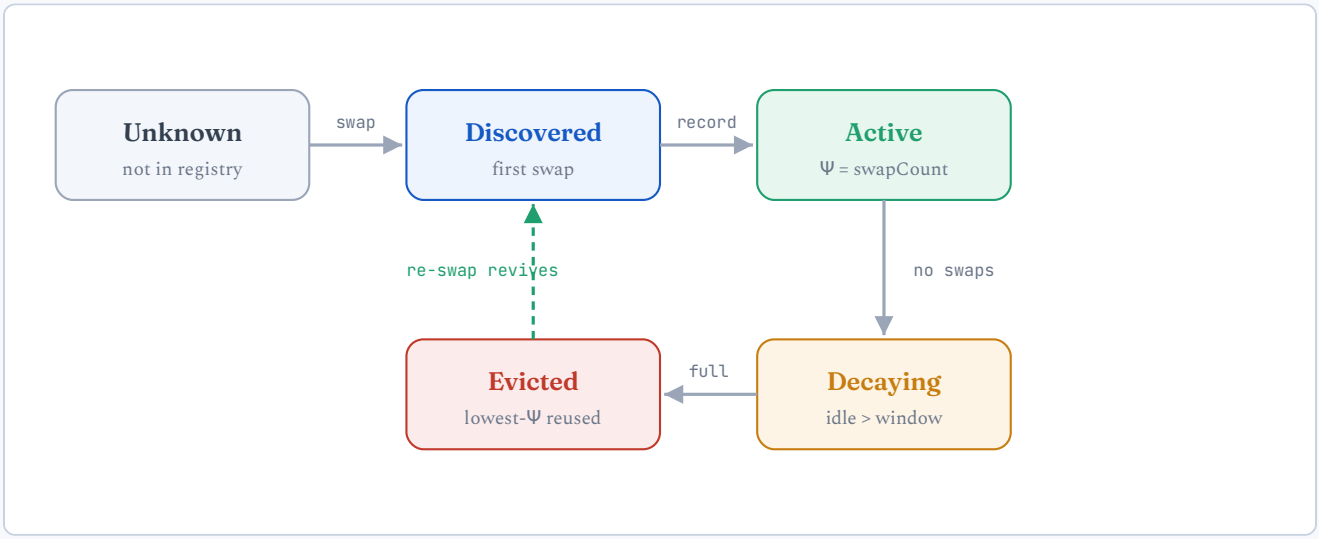

FIG. 8 *The registry lifecycle. A pool moves from unknown to discovered on its first successful swap, becomes active with vitality equal to its swap count, and either stays active through continued use or decays toward eviction through silence. Re-swapping revives a decaying pool; a deeper newcomer can evict a silent incumbent. No keeper, no cron, no off-chain process.*

10.2 Vitality-ranked eviction

Each token pair holds at most sixteen pools. When a seventeenth wants in, the Hub scans the incumbents, finds the one of lowest current vitality field Ψ — not raw activity, but the full Ψ that already folds in depth, bridge status and concentration — and overwrites it, but only if the newcomer's projected Ψ strictly exceeds the weakest occupant by a clear margin:

EVICTION ADMISSION RULE

$$\Psi_{\text{new}} \;>\; \Psi_{\min} \;+\; \tfrac{1}{4}\,\Psi_{\min} \;=\; 1.25\,\Psi_{\min} \tag{15}$$

Ranking by Ψ rather than by the bare activity score is what lets a deep newcomer displace a shallow-but-recently-active incumbent: depth enters the comparison directly, instead of being left to assert itself "over time." The 25% margin is deliberately strict: a newcomer must be materially healthier than the weakest incumbent, not a hair above it, so admission is decisive rather than a knife-edge, and an attacker cannot churn the registry with pools that are only marginally better. Equally, it closes a griefing vector — under a bare "displace the lowest" rule, an adversary could keep a deep competitor out by holding sixteen shallow slots barely active; ranking by Ψ with a margin means a genuinely deeper pool still wins. The combination of temporal decay (§8.3) and Ψ-ranked eviction means the sixteen slots converge, without supervision, on the sixteen pools that are simultaneously the deepest and the most used — precisely the set a router should consider.

10.3 Permissionless discovery

Discovery is a public, permissionless view. For any pair, the Hub iterates its factory list, derives candidate addresses by the modes of §6, and returns those that carry bytecode. The Solver merges this with the registered set and deduplicates by address, so neither source can suppress the other. To bound gas, the Solver applies a *freshness gate*: when at least three registered venues for a pair have seen activity within the discovery window — one hour of wall-clock time, measured by `block.timestamp` and therefore identical on every chain — it trusts the registry and skips the full CREATE2 sweep. New, thin or quiet pairs always fall through to discovery, so freshly-deployed pools are still found. The window is a gas/coverage knob, never a safety parameter: the on-chain floor protects every fill regardless of how stale the registry is.

Administrators retain a privileged `seedPool` entry-point to pre-populate the registry at deployment, but it is an optimisation, not a dependency: an empty registry heals itself into a populated one through ordinary trading. There is no indexer to run, no list to maintain, and no moment at which the protocol is blind because a server is down.

10.4 The coherence guard

The configuration surface is itself guarded. When an operator registers a venue factory, a coherence check rejects every structurally impossible combination before any state is written: an invalid kind; a CREATE2 mode with a zero init-code hash; a clone-mode slot bound to a non-Solidly kind; a V3-salt slot bound to anything but V3 or Algebra; an Algebra venue declaring a non-zero fee (its fee is dynamic, so its salt omits it). Each violation reverts early. A misconfigured venue can never silently derive wrong addresses, because a misconfiguration cannot be stored in the first place. Even were one stored, the `hasCode` guard and the floor would catch it — but the protocol prefers to fail at the door rather than rely on later defences.

11 Routing & the Split Allocator

With Ω, $\mathcal{G}$, Φ, Ψ and a self-curating registry in hand, the Solver evaluates the meta-equation over a deliberately small route space. Two topologies suffice to capture the overwhelming majority of realisable value, and the Solver computes all of them and returns the maximiser — plus the runner-up as a fallback, so a caller always has a second route in hand.

11.1 Two topologies

A **direct** route swaps $t_{\text{in}} \to t_{\text{out}}$ in one hop, splitting the input across as many as five parallel pools. A **bridge** route passes through a canonical intermediary — WETH or USDC — when no single venue holds the pair deeply enough; the global five-leg budget is strict and is partitioned between the two stages, typically three legs into the bridge and two out. The Solver evaluates the direct route and one route through each registered bridge, scores all of them under equation (1), and returns the highest. When a direct pool is healthiest it wins; when composition pays it wins instead. The choice is made per trade, not fixed in advance.

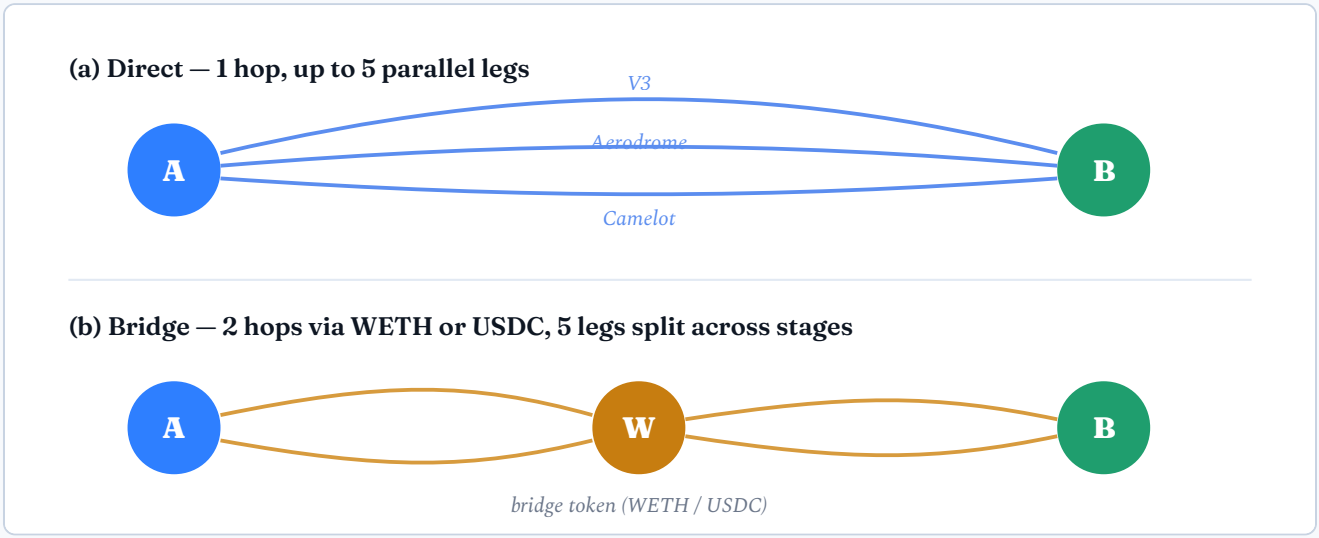

FIG. 9 *The two route topologies. (a) A direct route splits a single hop across up to five parallel pools. (b) A bridge route passes through WETH or USDC, splitting the global five-leg budget across two stages. The Solver evaluates both, and one route through each bridge, and returns the maximiser of equation (1).*

11.2 The capital-anchored filter

Allocating a split naïvely on full-input rates conflates two signals — price *quality* and pool *depth* — because a small but healthy pool shows a poor full-input rate purely from its own impact. Filtering on that rate discards good small pools when the trade is large, leaving the split with a single leg: precisely the failure mode observed in volume stress tests. BlazePhoenix instead filters in two clean stages. First it quotes each candidate with a *probe* of $x/100$ — small enough that the marginal rate is essentially spot price — and keeps only those whose marginal rate sits within a tight band of a reference rate:

MARGINAL-RATE QUALITY FILTER

$$\text{keep } i \iff R_i \in \left[\, R_{\text{anchor}}\left(1 - \tfrac{200}{10^4}\right),\;\; R_{\text{anchor}}\left(1 + \tfrac{200}{10^4}\right) \right], \qquad R_i = \frac{\Omega(\text{pool}_i,\, x/100)}{x/100} \tag{16}$$

The reference is not the plain median. A one-pool-one-vote median is manipulable: an adversary can deploy a cluster of dust pools quoting a stale, coordinated price and out-vote the single honest deep pool, which the filter then discards as an outlier — a free griefing vector that misroutes flow and, in the limit, steers it into pools that cannot fill it. (In one real case, two abandoned SushiV3 pools agreeing on a stale ~910 rate excluded a 401k-USDC pool quoting the true ~1633 as an "outlier.") Concentrated liquidity compounds the problem, since active liquidity is free to inflate with a hair-thin position and so cannot serve as a depth proxy either. BlazePhoenix anchors the band on the rate of the candidate holding the largest *real* balance of the output token, read by a single static call. Faking the anchor requires depositing more genuine capital than the honest deep pool — at which point the attacker *is* the deep pool, and arbitrage disciplines its price. The plain median is retained only as a fallback when no balance anchor exists, as for V4, whose currencies live in the manager singleton rather than in per-pool balances.

TRUTH VOTES WITH CAPITAL

Existence is cheap; capital is not. A dust pool can quote any price it likes, but it cannot hold the deepest real balance of the output token without becoming the very pool arbitrage keeps honest. Anchoring the quality band on capital rather than on a count of opinions makes the filter unforgeable without putting real money at the same risk an honest deep pool already bears.

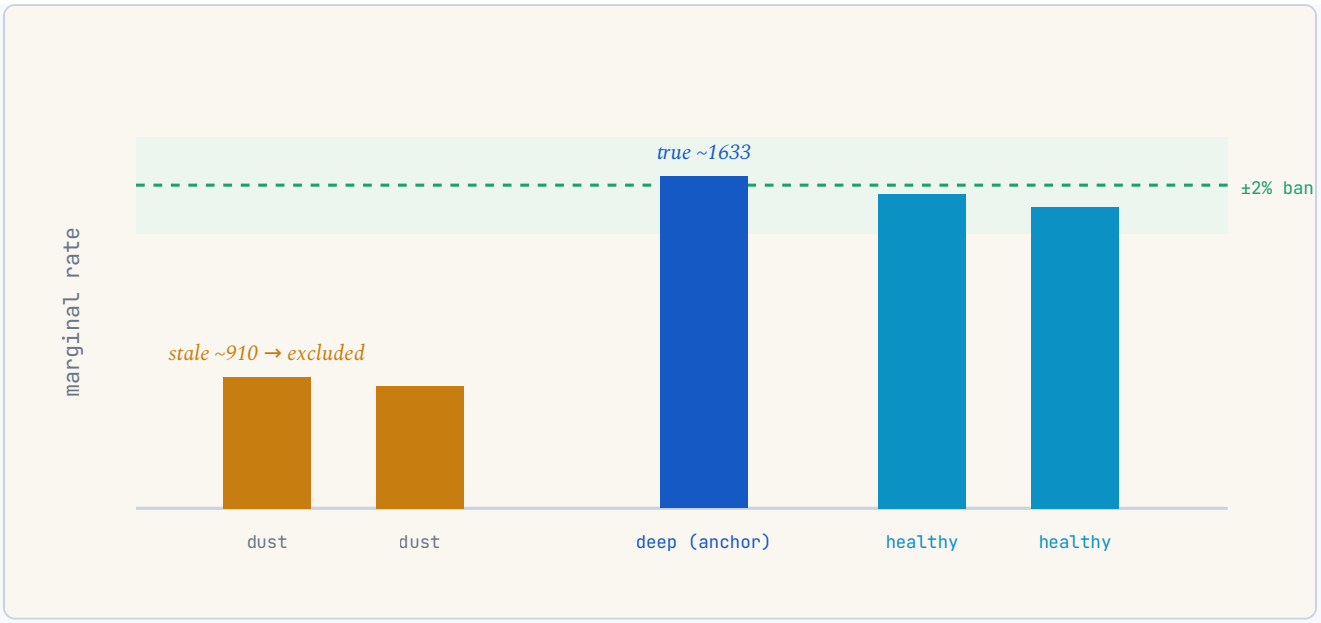

FIG. 10 *The capital anchor. Two dust pools agree on a stale price and would carry a one-pool-one-vote median; the anchor instead sits on the pool with the largest real output-token balance, so the stale cluster falls outside the ±2% band and is excluded, while genuinely healthy pools survive.*

11.3 Depth-weighted allocation

The surviving pools are then allocated by capacity, so a small but well-priced pool is kept but receives proportionally less. Allocating evenly is catastrophic: a uniform split across a 790-WETH pool and a 0.34-WETH dust pool destroyed roughly 20% of value in a measured WETH/USDC test (a 50/50 split returned 1302 USDC where concentrating in the deep pool returned 1638). The Solver instead weights each pool by its depth proxy, normalised to the range $[1, 10^4]$ *within its own kind* — because concentrated-liquidity L, constant-product reserves and Curve quoted-out are different units and cannot be compared across kinds. A dust pool then carries negligible weight and a deep pool carries near-full weight, and the depth-weighted allocation converges on the optimum (1638, matching deep-only). This is the correct division of labour: **filter on quality, anchored to capital; allocate on depth, normalised within kind.**

For routes of three or more legs, the Solver additionally shaves each leg's expected output by five basis points per leg, absorbing inter-leg drift on longer routes. Empirically the split never underperforms the best single venue — the protocol's standing guarantee is that a split is provably never worse than concentrating the whole trade in the single deepest pool.

12 The Execution Layer

The Solver produces a route; the Router executes it atomically or reverts. The Router is the only contract that moves funds, and it is built around a single principle: one mechanism per concern, applied universally. One callback handles every V3-shaped venue, one unlock dance handles every V4 singleton, one entry-point family handles every authorisation scheme. The contract walks the route's hops in sequence and dispatches each leg to the venue-appropriate primitive selected by its kind — the geometric counterpart of the dispatcher's pricing branch. The critical correctness property is that the Router rescales each stage's legs against the *actual* balance received, not the quoted balance, because a bridge route's second stage must operate on whatever the first stage truly produced.

> ONE MECHANISM PER CONCERN
>
> Every venue family is priced from its own arithmetic and settled by its own primitive: constant-product and stable curves by closed form, concentrated pools by revert-extraction (§12.4), Curve by its on-chain `get_dy` / `exchange`. No path in the system lets a number computed off the execution path govern a fill.

12.1 The One-Callback Doctrine

Every Uniswap-V3-shaped venue — V3 itself, Algebra, the concentrated Solidly forks, PancakeSwap V3, SushiSwap V3, and the rest — uses the same flash-accounting pattern: the pool calls back into the swapper mid-swap, demanding payment for the output it is about to release. Each fork names this callback differently, but the shape is identical: a delta for token0, a delta for token1, and arbitrary calldata. Rather than implement a named callback per fork, the Router exposes a single `fallback` that handles them all. It reads the two deltas, identifies which token it owes from their sign, and pays exactly that amount — reading the in-flight leg context (the committed pool, the input token, and the maximum it may spend) from transient storage so the callback executes without dirtying persistent state.

Because the callback is reachable by any caller, it is guarded on three fronts: it pays only the pool committed for the current leg; it refuses to act when no leg is in flight (the transient context is zero between swaps); and it never releases more than the leg's recorded budget. One function, every V3-shaped DEX, and no path by which an arbitrary caller can drain funds in flight.

12.2 Authorisation schemes

A trader must authorise the Router to move their input token. The protocol supports three schemes through distinct entry-points, so a user is never forced into an approval pattern their wallet cannot produce.

TABLE 9 — AUTHORISATION SCHEMES

Scheme	Mechanism	Trader benefit
Classic	`approve` + `transferFrom`	Universal; works with any ERC-20.
Permit2	Signature-based transfer, zero standing allowance	One-time setup, then gasless approvals across all integrations.
EIP-7702	Delegated account code	An EOA executes as a contract for the swap's duration; batched auth-and-swap.

12.3 The V4 unlock dance

Uniswap V4 replaces per-pool contracts with a single `PoolManager` singleton and an unlock-and-settle accounting model: the swapper unlocks the manager, performs swaps that accrue deltas, then settles every owed currency and takes every owed output before re-locking. The Router encapsulates this entire dance behind the same hop interface as every other venue, carrying the in/out currencies across the unlock callback in transient storage. After the swap, the manager returns a packed `BalanceDelta` — `amount0` in the high 128 bits, `amount1` in the low 128 — which the Router unpacks to determine what it owes and what it is owed, then settles via `sync → settle → take`. That the delta is *packed* rather than returned as two words was a detail visible only on a real fork, not in a mock; the protocol's validation discipline (§17.4) is what surfaced it. Before any V4 leg, the Router reads the hook address and rejects it outright if its bits declare delta-altering permissions (§15.1). To the Solver and to the trader, a V4 leg is indistinguishable from any other.

12.4 The Exact Pass

The Quoter previews a route off-chain before a trader commits to it. For constant-product, stable-swap, Solidly and Curve legs the closed forms of §4 are exact, and the Quoter evaluates them directly. For concentrated-liquidity legs — V3, Algebra, V4 — a closed form is exact only within a tick, so the binding quote is taken from the pool itself. The mechanism is **revert-extraction**: the Quoter calls the pool's real swap entry-point; the pool calls back demanding payment; instead of paying, the Quoter's callback reverts, carrying the computed output deltas in the revert payload. The revert unwinds all state — nothing is paid, nothing persists, the call is free of side effects by

construction — and the Quoter decodes the exact output from the bytes it caught. V4 is handled identically through an unlock whose callback reverts with the packed balance delta. There is no replicated tick arithmetic anywhere in the quote path.

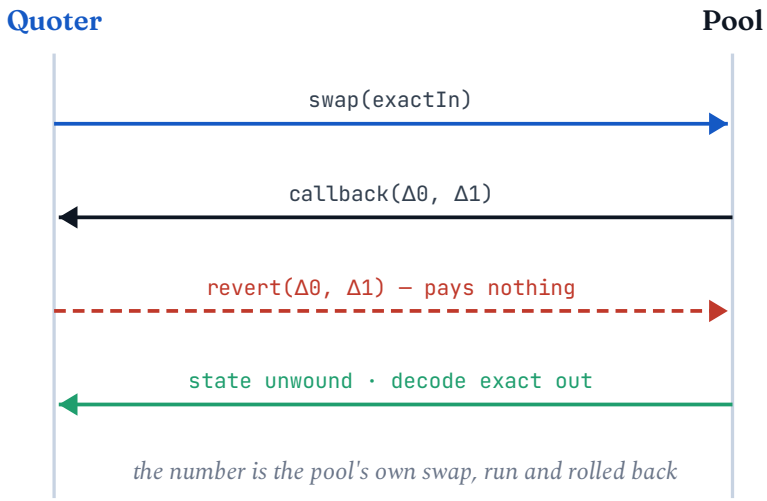

FIG. 11 *The Exact Pass. The Quoter runs the pool's real swap and reverts from inside the callback, carrying the deltas in the revert payload. The revert unwinds all state, so the call is side-effect-free, and the decoded output is the venue's own bytecode over live state — it cannot diverge from an execution at the same block, because it is that execution, rolled back.*

Off-chain the Quoter reports a net receive that already accounts for the protocol fee and a small inter-leg safety buffer:

QUOTER VALUATION

$$y_{\text{net}} \;=\; y_{\text{gross}} \cdot \left(1 - \frac{28}{10^4}\right) \cdot \left(1 - \frac{s(\ell)}{10^4}\right), \qquad s(\ell) = \min\big(10,\ \max(0, \ell - 2)\big) \tag{17}$$

where y_{gross} is the Solver's output (already net of pool fees), the protocol fee is 28 bps applied to the quoted output only, and $s(\ell)$ is a dynamic buffer — zero at two legs or fewer, one basis point per additional leg, capped at ten — absorbing inter-leg drift on longer routes. A route is unquotable (returns zero) if it touches a pool whose hook is delta-altering or not allow-listed, the anti-scam gate of §15.1.

QUOTE-EXECUTION COHERENCE

Validation measured the realised on-chain output against the exact-pass quote on every routed pair across four chains. The two agreed to a fixed, deterministic divergence equal to the protocol fee on the quoted output — that is, the pre-fee quote matched the realised swap to within rounding. The single mechanism that prices a concentrated pool off-chain is the pool's own swap, run and reverted.

12.5 Fee-on-transfer and rebasing

A fee-on-transfer token delivers less than its nominal amount on every transfer, and some apply the fee through a hook that itself trades on the same pool mid-transfer; a rebasing token can deliver a few units short through share-rounding. Either breaks any accounting that assumes `amount sent == amount received`. The Router handles both on constant-product legs by the canonical supporting-fee pattern: it transfers first, then reads the pool's reserves and its balance in the *same* post-transfer state, and prices the output on the difference the pool actually sees. That quantity is correct by construction regardless of what a transfer hook did in between, because it is exactly what the pool's own invariant will check — the same *ask, never replicate* doctrine as the Curve adapter and the Exact Pass. (One real token, FLOKI, trades on the very pair being swapped during its transfer; the post-transfer recompute prices it correctly anyway.)

The branch engages only when a token is measured to deliver short, so honest tokens follow the original path unchanged. When it engages, the formula-derived floor — blind to the transfer fee, and therefore inflated — is dropped for that swap; the trader's own minimum-output bound and the protocol floor on the *real* output continue to protect them, and slippage is ultimately enforced on the amount the recipient actually receives, not the nominal figure. Concentrated-liquidity legs do not support fee-on-transfer by design — the flash-accounting callback must pay the exact amount owed — so the protocol confines fee-on-transfer handling to the constant-product legs where it is well-defined.

13 Fee Policy & the Surplus Rule

The protocol charges a fee of 0.28% — 28 basis points — on the quoted output of a swap, split between two treasuries:

PROTOCOL FEE AND TREASURY SPLIT

$$\text{fee} = \frac{28}{10^4}\, y_{\text{quote}}, \qquad \text{treasury}_1 = \frac{30}{100}\,\text{fee}, \qquad \text{treasury}_2 = \frac{70}{100}\,\text{fee} \tag{18}$$

The design decision worth stating is what the fee is charged *on*. The fee applies only to the output the protocol quoted. Any surplus — output realised above the quote, whether from favourable rounding, a price move between quote and execution, or the deliberate under-quoting margin of §5 — is paid entirely to the trader, fee-exempt by policy.

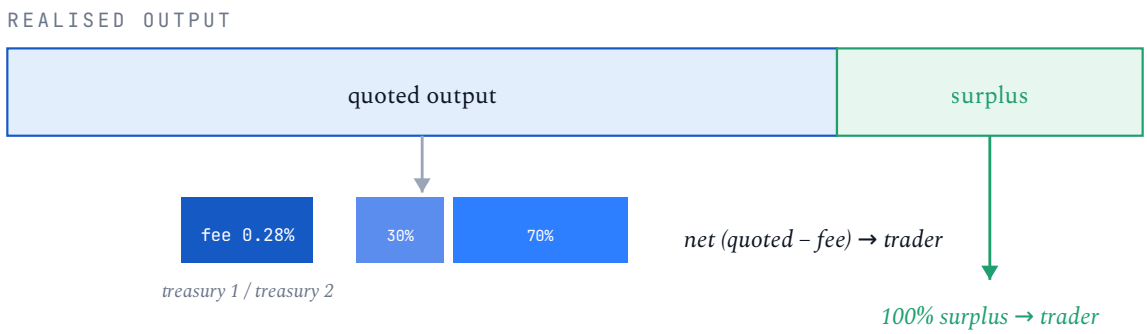

FIG. 12 *The fee & surplus waterfall. The 0.28% fee is charged on the quoted output only and split 30/70 between two treasuries; the net and the entire surplus flow to the trader. The protocol never profits from the gap between what it promised and what it delivered.*

THE SURPLUS RULE

The protocol never profits from the gap between what it promised and what it delivered. A trader keeps 100% of any positive surplus. This aligns the protocol with the trader on the one axis where aggregators are most tempted to skim — the difference between quote and fill — and makes the headline fee the true and only fee.

One subtlety governs how these quantities reach the Router. The route carries advisory accounting fields, but the Router does not trust them for either the fee base or the floor. It re-derives both on-chain: it measures each leg's real price impact from live reserves as the swap executes, recomputes the floor from that measured impact and the true leg count, and applies the floor to the realised output. The fee base is $\min(\text{attested quote},\ \text{realised})$, floored at the protocol-floor fraction of the realised output — so a caller cannot drive the fee toward zero by understating the quoted total, and cannot inflate it beyond what was received. The surplus exemption is preserved — output above the attested quote is still returned fee-exempt — but the protocol's revenue and its safety floor are now computed properties of the actual trade, not numbers the caller could choose. The fee is honest by construction.

14 Arithmetic Safety

Every formula in §4–§5 lives inside the EVM's 256-bit unsigned integer arithmetic, where an overflow is not a rounding error but a reverting panic. The Core proves, rather than assumes, that each intermediate fits — because on an immutable ledger an unproven bound is a latent, unbounded liability.

Concentrated-liquidity products.

The dangerous term in the V3 quote is $L \cdot \sqrt{P}$, which can reach 2^{288} — thirty-two bits beyond the word. The Core never forms it. Every product passes through a full-precision `mulDiv` primitive that computes $a \cdot b/d$ with a 512-bit intermediate and a 256-bit result, so the value is bounded even when the naïve product is not. Each branch of the V3 quote is a chain of such calls whose every output provably fits in 256 bits.

Quartic cubic terms.

The Solidly invariant forms x^3 and y^3. The Core bounds each normalised reserve to 3.4×10^{38} ($\approx 2^{128}$) before the solve and returns zero — treating the pool as unquotable — for any state exceeding it. An economically nonsensical swap that would overflow is declined, never panicked; the protocol fails closed rather than reverting in a way an attacker could weaponise.

Newton denominators.

Both the stable-swap and Solidly solvers guard their denominators: a non-positive Newton denominator means the step is undefined for that state, and the solver bails to zero rather than divide by it. The Solidly step additionally clamps a downward move to at most half the current iterate, so y provably never crosses zero.

TABLE 10 — OVERFLOW GUARDS BY FORMULA

Computation	Risk	Guard
V3 product $L \cdot \sqrt{P}$	2^{288} overflow	512-bit `mulDiv` intermediate
Solidly cubic x^3, y^3	overflow	reserve cap $3.4 \times 10^{38} \to 0$
Newton step f/f'	div-by-zero	non-positive denominator → 0
Solidly down-step	cross zero	half-step clamp
Price impact	>100%	ceiling at 10^4 bps

The configuration surface is guarded too, by the coherence check of §10.4: every structurally impossible venue registration reverts before any state is written. Across the pricing primitives, the address derivation and the configuration guards, the protocol's posture is uniform — where a bound can be proven it is proven, and where it cannot be guaranteed the operation halts rather than proceeds on a guess.

15 Security Model

The protocol's trust surface is deliberately narrow. The Router holds no funds between transactions; every swap is atomic and the Router's balance at rest is zero. We enumerate the principal threats and the mechanism that addresses each.

TABLE 11 — THREAT MODEL

Threat	Mechanism
Adverse price movement / sandwich	User minimum-output bound; the transaction reverts rather than fill below it.
Quote–execution drift	Same function, same state, same block on both sides; dynamic fees read live; concentrated quotes by revert-extraction.
Malicious or wrong pool address	CREATE2 derivation verified against ground truth; `hasCode` guard discards any non-existent derived address.
Misconfigured venue	Configuration-coherence guard reverts structurally impossible registrations before any write.
Callback spoofing	Callback authorises the caller against the in-flight leg context in transient storage, and rejects any pool demanding more input than the leg intended to spend.
Untrusted route accounting	Fee base and floor re-derived on-chain from measured impact and realised output; caller fields can tighten but never relax protection.
Reentrancy	Reentrancy guard on execution entry-points; transient state cleared per swap.
Hostile hook (V4)	Three independent gates: address-bit inspection, an administrator allow-list, and the anti-scam projector. See §15.1.
Arithmetic overflow	Proven bounds and 512-bit factoring throughout (§14).

15.1 Defending against hostile V4 hooks

Uniswap V4 lets any pool attach a hook — arbitrary code that runs at defined points in a swap's lifecycle. This is the most dangerous primitive an aggregator must integrate, because a hook executes inside the swap the Router is settling. A hostile hook's goal is to alter the accounting

between the moment the Router commits to pay and the moment it is paid. The naïve defence — calling the hook to ask what it will do — is no defence at all: the hook lies, or reenters. **BlazePhoenix never asks the hook anything.** It defends on three independent layers, any one of which is sufficient, arranged so the cheapest and most certain runs first.

HOOK DELTA-PERMISSION TEST — A SINGLE AND, NO EXTERNAL CALL

$$\text{altersDeltas}(h) \;=\; \Big(\big(\text{uint160}(h)\;\&\;\texttt{0x3FFF}\big)\;\&\;\big(2^3 \mid 2^2\big)\Big) \neq 0 \tag{19}$$

Layer 1 — the permission bits are in the address, and the address cannot lie. V4 encodes a hook's permissions in the low 14 bits of its own contract address. Those bits are not a value the hook reports; they are fixed at deployment by the CREATE2 salt and are physically immutable — the hook cannot have a permission its address does not declare, because the manager checks the same bits before invoking it. Two of those bits, `BEFORE_SWAP_RETURNS_DELTA` and `AFTER_SWAP_RETURNS_DELTA`, are the only ones that let a hook modify swap accounting. Equation (19) masks them with a single `AND`; if either is set, the leg is rejected before a single token moves. It costs one bitmask, makes no call into untrusted code, and cannot be evaded.

Layer 2 — an explicit allow-list for everything else. A hook without delta permissions can still revert maliciously to grief or consume gas, so a V4 pool is routable only if its hook is additionally on an administrator-curated allow-list. The default is closed: an unknown hook is not routed, even if its bits look benign. The Hub enforces this at registration — a V4 entry with a non-allow-listed hook reverts — so a hostile hook never reaches the Solver's candidate set.

Layer 3 — the quote refuses to surface the route at all. The anti-scam projector values any route touching a delta-altering or non-allow-listed hook at exactly zero, so such a route is never even shown to the trader as a candidate:

ANTI-SCAM PROJECTOR Ξ

$$\Xi(\text{route}) \;=\; \prod_{\text{legs } i} \mathbb{1}\big[\,\text{hook}_i \text{ allow-listed} \;\wedge\; \neg\,\text{altersDeltas}(\text{hook}_i)\,\big] \tag{20}$$

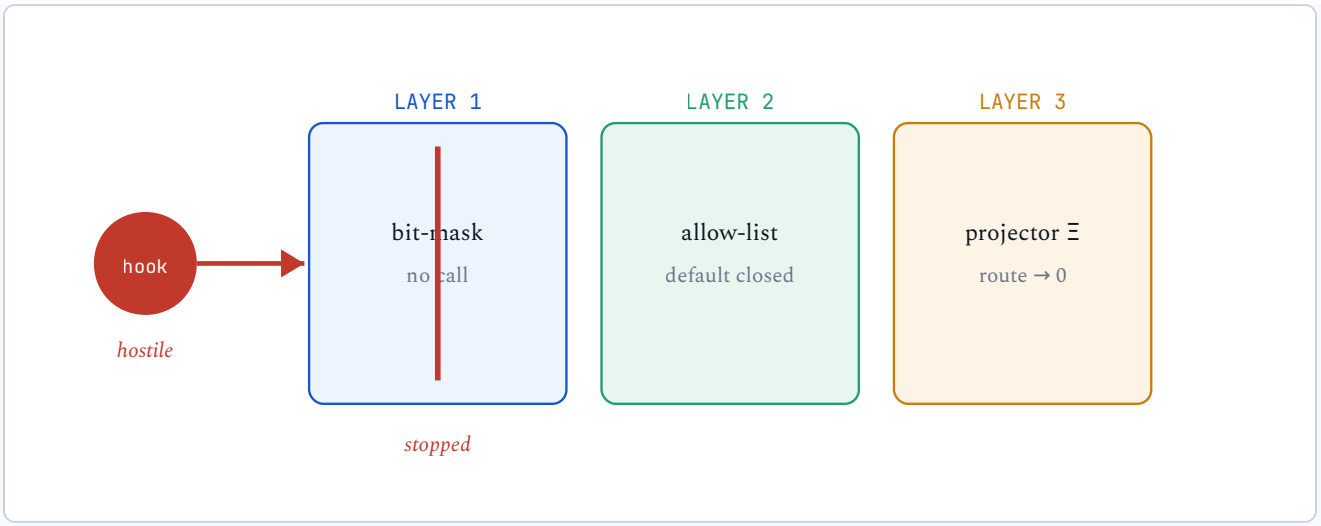

FIG. 13 *Three redundant gates against hostile V4 hooks. Bit-inspection makes the accounting attack impossible without any call into untrusted code; the allow-list makes the unknown case non-default; the projector makes the bad route invisible to selection. Defeating the protocol would require a hook that simultaneously carries no delta bits, sits on the allow-list, and yet still subverts settlement — a combination the first layer's appeal to V4's own immutable enforcement rules out.*

15.2 Administrative powers and the One-Way Door

Administrative authority is split into two tiers with sharply different danger profiles. **Curator** powers can only grow the registry — list a factory, add a bridge token, allow a hook — and can never touch funds or redirect flow; a malicious listing cannot drain, because every pool is validated at quote and execution and bounded by the floor and the user's own minimum-output. **Control** powers are the dangerous ones: they can set contract roles, toggle the operator set, pause the protocol, change the V4 manager, remove a bridge, redirect the treasuries, or transfer admin. The administrator can do these things — until it chooses, irreversibly, that it no longer can.

Both the Hub and the Router expose a `renounceControl` entry-point that permanently disables every Control power while leaving the grow-only Curator powers intact. After it is called, the treasuries, the Permit2 address, the pause flag, the role wiring and the admin key are frozen at their current values forever; the contracts keep executing swaps under that fixed configuration, and new venues can still be listed, but no privileged party can ever redirect or freeze the protocol again. It is a one-way door: reversible only until the moment it is walked through, and never afterward.

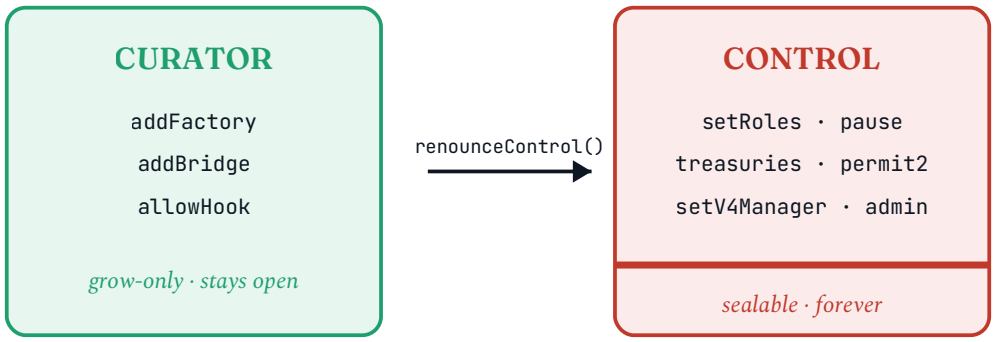

FIG. 14 *The One-Way Door. A single call permanently seals every fund-touching and freeze power while the grow-only registry powers remain open. Autonomy becomes a property of the contract, not a promise in a document.*

WHY THE DOOR IS THE POINT

A protocol can promise it will never abuse an admin key, but a promise is not a guarantee — a compromised key keeps every power the contract grants it. The One-Way Door converts the promise into mathematics: once walked through, the dangerous powers do not exist, for anyone, ever. What remains is a contract that can be extended but never captured — the difference between “we won't” and “we can't.”

✦Why Web 2.5 Crypto Fails Its Users

The aggregator space is crowded. But most of it is a theatre. Behind the on-chain interfaces live fleets of servers, proprietary solvers, liquidity-indexing bots, and administrator keys that can rewrite the rules mid-game. The chain handles settlement; the internet handles everything that matters. This is not decentralisation — it is the centralised web wearing a blockchain mask, and its failure modes are the failure modes of the centralised web, inherited wholesale.

These are not hypothetical risks. They are the mechanics that have, repeatedly, drained real value from real users — and that any honest protocol comparison must name directly. We name seven of them.

$320M+	$200M+	∞	0
DRAINED BY ORACLE ATTACKS 2022–24	LOST TO UPGRADEABLE-PROXY EXPLOITS	ADMIN KEYS THAT COULD REDIRECT FUNDS	OF THE ABOVE APPLY HERE

The seven structural failures — and what actually closes them

1 · QUOTE-FILL

Quote–fill divergence

Web 2.5: Off-chain solver quotes X; on-chain fills X–ε. Gap is unverifiable, routinely skimmed. MEV bots front-run the published route between quote broadcast and block inclusion.

BlazePhoenix: The Exact Pass runs the pool's own swap bytecode and reverts — the number shown IS the number filled, same bytecode, same live state. Divergence is a structural impossibility.

2 · STALE GRAPH

Stale liquidity graph

Web 2.5: Keeper last updated N seconds ago. Router picks a dead or depleted pool. During congestion, keeper lag grows exactly when accurate routing matters most — failed swaps, adverse fills, no recourse.

BlazePhoenix: The Self-Healing Registry learns every pool by executing it. There is no keeper to lag, no list to push, no moment at which the protocol is blind because a server is down.

3 · CENSORSHIP

RFQ censorship and liveness failure

Web 2.5: RFQ server silently blacklists an address, token, or geography. No quote, no explanation. Infrastructure outage equals protocol outage. Users have no fallback and no proof of what happened.

BlazePhoenix: No RFQ server exists. The Quoter is a public, permissionless view function on a stateless Core — it cannot censor any address, and it cannot go offline while the chain runs.

4 · UPGRADE RUG

Admin upgrade / rug vector

Web 2.5: Proxy admin replaces audited logic with extractive logic in one transaction. "Audited" applies at audit time, not at upgrade time. A 2-of-3 multisig can collude. This has happened to named protocols with $200M+ in losses.

BlazePhoenix: No proxy. No upgrade path. The One-Way Door (§15.2) permanently burns every control-tier power. The function that would redirect funds does not exist in the deployed bytecode — not disabled, absent.

5 · MEV ROUTE

MEV from route publication

Web 2.5: Route is an off-chain artifact submitted as known calldata. Searchers front-run the announced path, sandwich the fill, and extract the margin from the user before settlement. Private mempools are a partial mitigation that requires trusting the relay.

BlazePhoenix: Route computed on-chain from live reserves at execution time. There is no pre-announced route to front-run; the fill price is determined by the block itself.

6 · DUST ATTACK

Dust-pool median manipulation

Web 2.5: Attacker deploys 16 micro-pools at a coordinated stale price, out-votes the honest deep pool in a median filter, and causes flow to be misrouted into pools that cannot fill it. Curator whitelists are the fix — which is a trusted centralised dependency.

BlazePhoenix: The Capital Anchor (§11.2) sets the reference rate from the pool holding the largest real output-token balance. Faking it requires outweighing the honest deep pool with real capital — at which point arbitrage disciplines it honestly.

7 · BYTECODE

Cross-chain bytecode drift

Web 2.5: Per-chain backends accumulate subtle fee, rounding and derivation differences. Same swap, different chains, different results. Each deployment needs a separate audit; a fix on one chain doesn't propagate. Users on different chains get a different protocol.

BlazePhoenix: One identical bytecode on every chain. CREATE2 derivation verified cross-chain against live factory state. A fix in the Core propagates to every deployment that imports it — one audit covers all four chains.

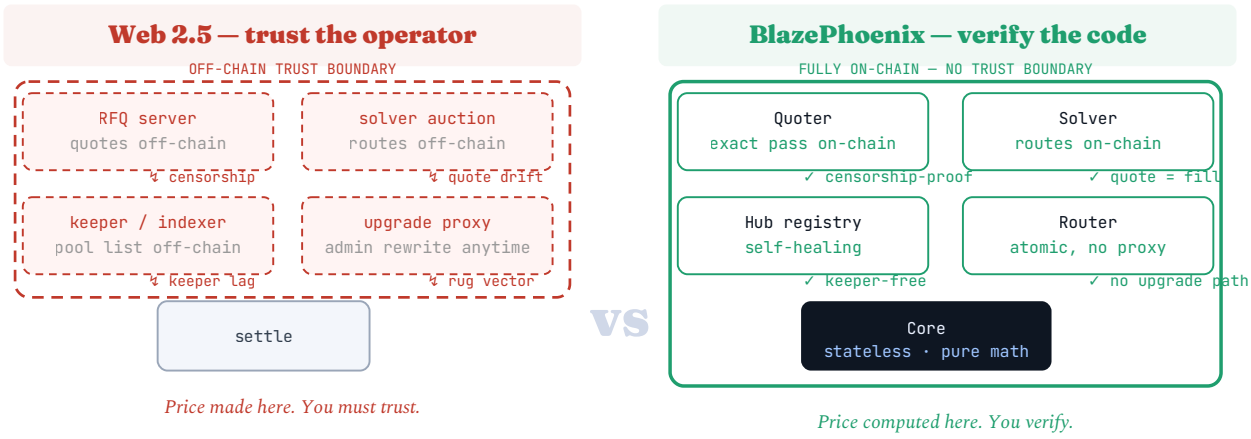

FIG. X *Two architectures for the same job. Every component above the settlement layer in Web 2.5 lives off-chain and is unverifiable — it can censor, lag, lie, or disappear. BlazePhoenix has no off-chain layer: pricing, routing, pool discovery and settlement all happen inside contracts that anyone can read. The trust boundary that Web 2.5 asks you to cross does not exist here.*

The asymmetry is permanent, not fixable

A Web 2.5 aggregator can add more audits, more multisig signers, more monitoring dashboards, more insurance — and still cannot close the gap. Because the gap is not operational; it is architectural. An off-chain solver that quotes prices will always have a window between quote and fill where the user cannot verify what happened. A keeper-maintained pool list will always have a staleness window between the last push and the current block. An upgradeable proxy will always have an admin who could, in one transaction, become adversarial. These are not bugs that can be patched; they are the *price of the off-chain escape hatch*, paid by the user, sometimes visibly as a revert and sometimes invisibly as a skimmed basis point.

The only answer to a structural problem is a structural solution. The structural solution is to move the decision on-chain — not to simulate on-chain behaviour with increasingly elaborate off-chain systems that are required to behave correctly and trusted to do so. BlazePhoenix pays a real price for this: concentrated-pool quoting costs more gas than reading a cached off-chain price; the self-healing registry must learn its pools by trading rather than ingesting a maintained list; routing is bounded by what is computable in a single transaction rather than what is optimal in a server farm. These are not embarrassments. They are the honest costs of a property that the alternative cannot offer at any price: **a protocol whose honesty is provable rather than promised**.

> THE PERMANENT ASYMMETRY
>
> A Web 2.5 aggregator can become more trustworthy. It cannot become trustless. BlazePhoenix begins trustless and stays there — because the properties that make it so (no off-chain solver, no keeper, no proxy, no upgradeable logic) are not policies that an admin chose and could change, but structural absences whose reinstatement would require a re-deployment the users would see. Trustless is not a higher point on the same scale as trustworthy; it is a different kind of thing entirely. Compute, do not trust.

16 Stateless · Serverless · Ungovernable

Three words describe what BlazePhoenix is, and together they describe what Web 3.0 was supposed to mean before the industry quietly compromised on each. The protocol is *stateless* at its core, *serverless* in its operation, and *ungovernable* in the only dimensions where governance is a weapon. These are not slogans bolted onto a conventional design; they are properties that fall out of the decisions made in every preceding section.

Stateless.

The Core is a pure library: no storage, no owner, no upgrade path — only functions. Pricing, address derivation, the floor and the vitality field are all pure computations over their arguments. Any contract, internal or external, that imports the Core gets byte-identical answers, and there is no instance state to diverge between callers, no admin to retune a constant, no proxy to swap the logic. The protocol's economic behaviour is a function in the mathematical sense — the same inputs always produce the same outputs, on every chain, forever.

Serverless.

There is no keeper, no indexer, no request-for-quote server, and no off-chain solver. Discovery is a permissionless on-chain view; the registry learns liquidity by trading and curates itself through vitality decay and Ψ-ranked eviction; quoting and routing are pure read-path functions any caller can run via `eth_call`; and the binding quote for a concentrated venue is the pool's own swap, extracted by revert. Nothing in the critical path depends on a process someone must keep running. There is no server to go dark, no database to fall stale, and no backend whose honesty the user must assume. The chain is not a settlement rail for decisions made elsewhere — the chain *is* where the decision is made.

Ungovernable.

Through the One-Way Door of §15.2, every power that could redirect or freeze the protocol can be permanently surrendered, leaving only the grow-only registry powers that cannot touch funds. After renunciation there is no key that can pause the Router, redirect a treasury, rewire the contracts, or seize a position — not because a multisig agreed not to, but because the functions that would do so no longer execute. Governance survives exactly where it is benign (listing new venues) and is abolished exactly where it is dangerous (touching money).

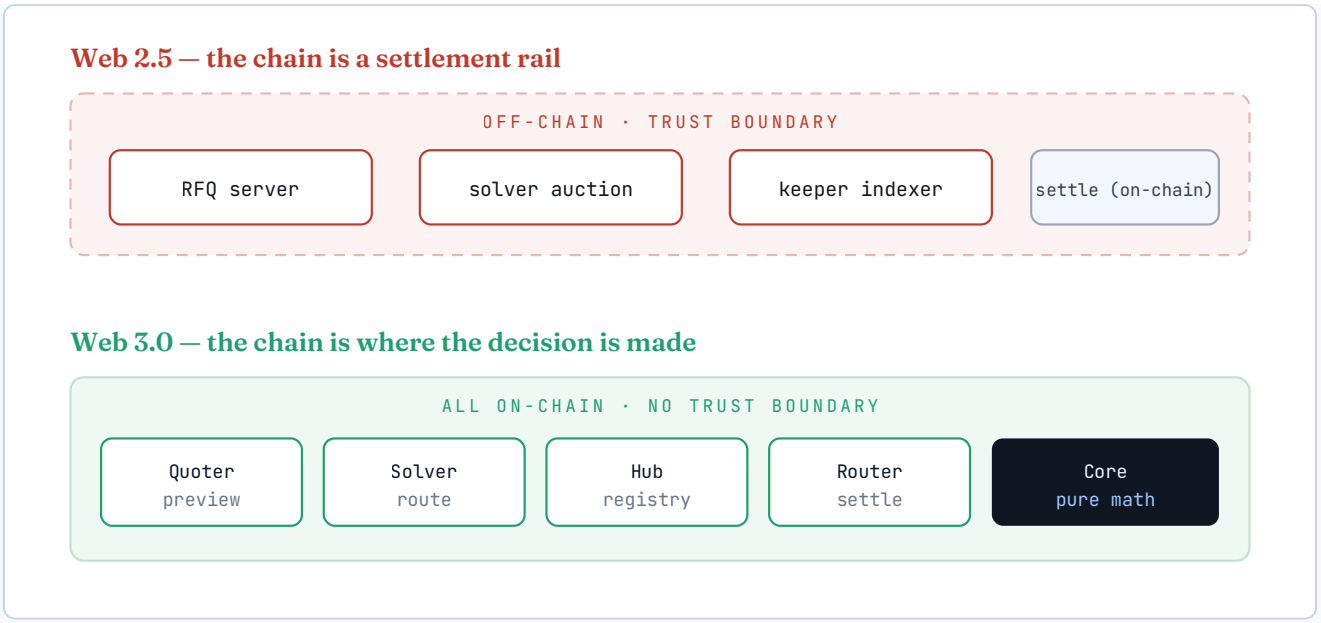

FIG. 15 *Two topologies for the same problem. In the Web 2.5 design the price is produced off-chain, behind a trust boundary the user cannot cross or verify, and the chain merely settles. In BlazePhoenix every stage — preview, routing, registry, settlement, and the math under all of them — lives on-chain, so there is no boundary across which a promise can quietly differ from a fill.*

> ***"A number quoted off-chain that equals the number realised on-chain, because they are computed by the same function over the same state. A registry that learns liquidity by trading it. A contract that can be extended but never captured. This is what it means for a protocol to live on a chain rather than merely settle on one."***
>
> THE BLAZEPHOENIX THESIS

17 Empirical Evaluation

BlazePhoenix was validated against forked mainnet state on Ethereum and three Layer-2 networks, alongside a unit and integration suite exercising the Core's pure mathematics and its behaviour against live liquidity. We report the headline results; all figures are from execution against real on-chain reserves at recent blocks, from a single identical bytecode.

17.1 Cross-chain coverage

The same contract bytecode was deployed and exercised on each chain, adapting to each network's distinct venue mix — Uniswap V4, Aerodrome and PancakeSwap on Base, Camelot on Arbitrum, Velodrome on Optimism — with no per-chain code change. Each chain wires at least five factories; Curve and Balancer-family venues are deliberately excluded from the validated registered set, pending their own fork-test coverage (§19).

TABLE 12 — CROSS-CHAIN VALIDATION, IDENTICAL BYTECODE

Chain	Factories	UniV3 discovery	V4	Concentrated quote
Ethereum	6 + V4	CREATE2 ✓	✓	revert-extraction ✓
Arbitrum	7	CREATE2 ✓	—	revert-extraction ✓
Optimism	6	CREATE2 ✓	—	revert-extraction ✓
Base	7 + V4	CREATE2 ✓	✓	revert-extraction ✓

17.2 Quote–execution coherence

On every routed pair the realised output matched the exact-pass quote to within the protocol fee — a fixed, deterministic divergence rather than slippage noise — including on the Algebra-based Camelot venue, where the universal callback and the revert-extraction quote operate unchanged. The Solver selected bridge intermediaries adaptively, routing some pairs through WETH, others through USDC, others direct, choosing the maximiser of equation (1) per pair rather than following a fixed rule. Concentrated venues that a single-tick quote had previously caused to be rejected at

the floor — and fee-on-transfer and rebasing tokens that had previously failed the return leg — route cleanly under the Exact Pass and the canonical fee-on-transfer path. A deliberately isolated pair returns no route rather than a fabricated one.

17.3 Gas profile

The protocol's caching design separates a cold first-discovery cost from a warm steady-state cost. On Base, a representative basket showed a consistent ~38% warm-cache saving.

TABLE 13 — COLD VS. WARM GAS, BASE MAINNET FORK

Token	Cold gas	Warm gas	Saved
USDC	461 030	280 367	39%
AERO	287 833	183 774	36%
cbETH	427 217	270 821	36%
BRETT	245 883	147 294	40%
TOSHI	361 117	206 955	42%
average	356 616	217 842	**38%**

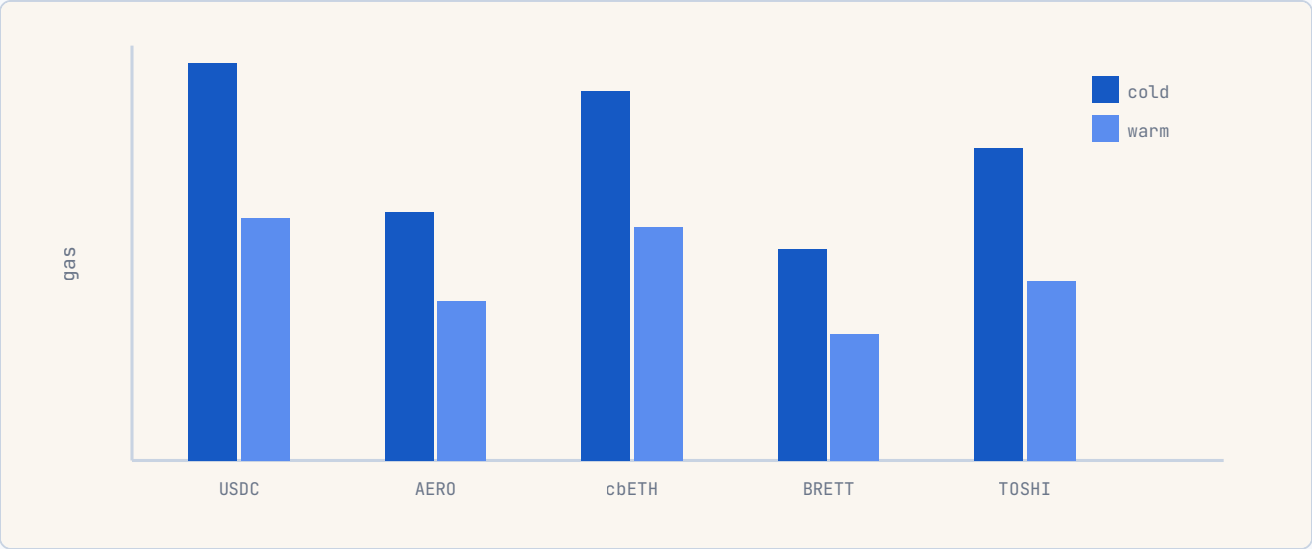

FIG. 16 *Cold vs. warm gas on a Base mainnet fork. The first discovery of a pair pays to learn it; every subsequent swap reads the cached Monoslot, for a consistent ~38% saving. On the Layer-2 networks the protocol targets, even the deepest multi-leg bridge route costs fractions of a US cent.*

17.4 Validation discipline

The validation suite divides into deterministic unit tests over the pure pricing primitives, address derivation and configuration guards, and fork-mode integration tests against live liquidity. Two principles govern it. Regressions are judged only on isolated runs — a clean fork per token — because sequential swaps against a shared fork accumulate state drift that distorts round-trip measurements. And concentrated-pool quotes are validated against the canonical venue quoters to the unit, in *both* directions, before integration: a path validated in one direction only can hide a defect in the other. A failing test is treated as a question to answer, not a number to force green — on one occasion a failing test pointed at a correct contract, and satisfying it would have introduced a bug. It was the test, not the contract, that was wrong.

18 Architecture

Five contracts, one of which is pure mathematics imported by the other four.

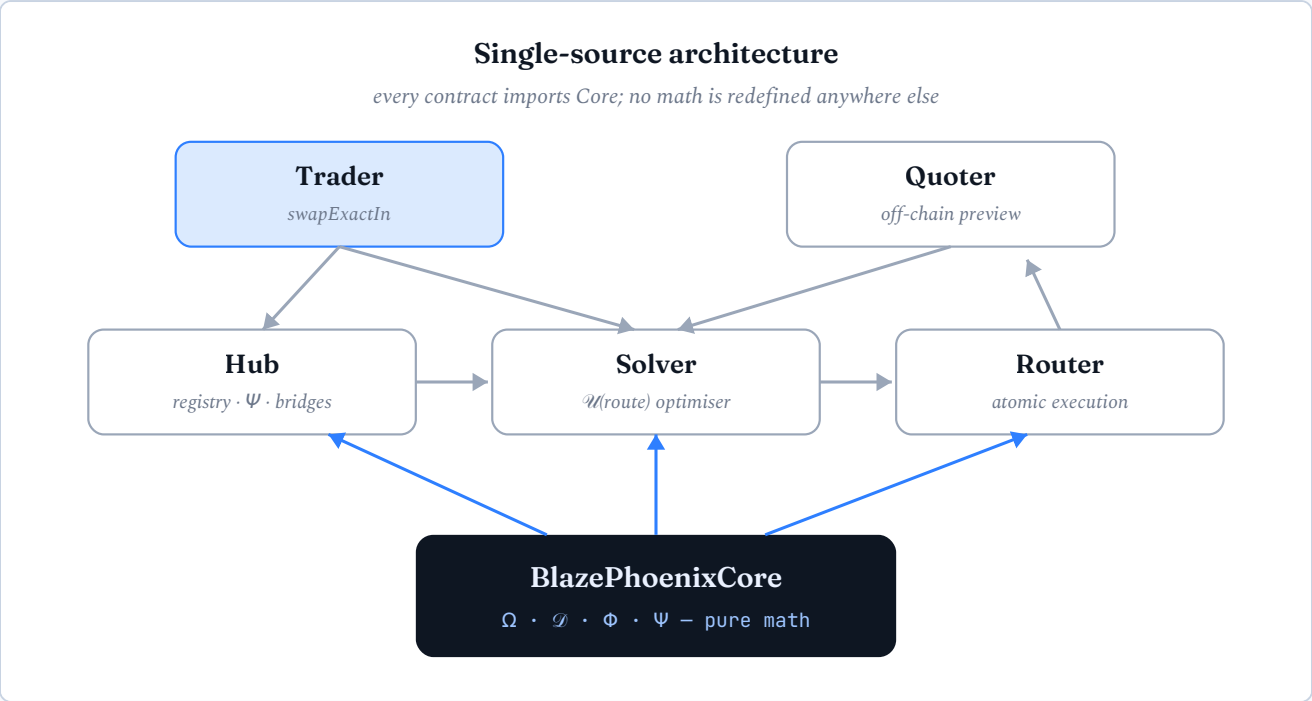

FIG. 17 *The single-source architecture. The Core holds every operator as pure functions; the Hub maintains the venue registry and vitality field; the Solver maximises the meta-equation; the Router executes atomically; the Quoter previews off-chain. No contract redefines a Core primitive.*

The **Core** is a stateless library: no storage, no owner, no upgrade path — only functions. The **Hub** holds the venue registry, the vitality field and the bridge-token set, guarded by the coherence checks of §10.4, with storage laid out under ERC-7201 namespacing so it can never collide with a consumer's slots. The **Solver** is a pure read-path optimiser that touches no state it can derive. The **Router** is the only contract that moves funds, and it moves them atomically under the user's minimum-output bound, holding nothing at rest. The **Quoter** mirrors the Solver for off-chain preview, by construction producing the same numbers the Router will realise. Each contract exposes a single compact error path — one custom error with a code — keeping the revert surface as legible as the happy path.

19 The Staking Engine

An aggregator routes other people's liquidity. A protocol must also cultivate its own. The BlazePhoenix Staking Engine turns the native token, BZPX, from a static balance into productive capital — simultaneously a yield instrument, a collateralised credit facility, and the alignment mechanism that binds long-term holders to the protocol's success. It is a single contract that does the work most protocols split across three, and it extends the aggregator's discipline to the protocol's own capital: *compute, do not trust*, now applied to solvency itself.

The design rests on one structural insight: do not build three systems where one suffices. A staking vault, a lending market, and a lock-boost incentive layer are usually separate contracts with separate accounting and separate attack surfaces. Here they are one accounting system with two yield accumulators over a shared stake base, and every privileged path is bounded by a proof rather than a promise. Above all, the contract is **provably solvent**: it stands behind a single conservation equation that the EVM itself enforces on every value-moving transaction, and it maintains itself with **no keeper**, **no oracle**, and **no administrator who can touch a staker's principal**.

> THE STAKING THESIS
>
> Staked BZPX is collateral, yield principal, and governance weight at once. A staker may simply earn emission; or borrow against their stake; or lock longer for a boosted share. The contract reconciles all three roles through a single `UserInfo` record and two MasterChef-style accumulators, with liquidation, interest and reward mathematics that are each closed-form, overflow-bounded, and — uniquely — subordinate to one solvency invariant that cannot be violated by any reachable state.

19.1 Emission and global parameters

The reward budget is fixed at genesis and emitted linearly. A total of 180,000,000 BZPX is released over a seven-year emission period, giving a constant emission rate:

LINEAR EMISSION RATE

$$\dot{R} = \frac{180{,}000{,}000 \cdot 10^{18}}{7 \times 365 \times 86400} \ \text{BZPX-wei per second} \tag{21}$$

Several constants bound the system against the failure modes that have historically drained staking contracts. They are stated once, in the contract, as immutables — and they are the entire tunable surface of the protocol.

TABLE 18 — STAKING ENGINE PARAMETERS

Parameter	Value	Purpose
Total rewards	180,000,000 BZPX	Fixed emission budget; no inflation beyond it.
Emission period	7 years	Linear release; no cliffs, no halvings.
Max stake / wallet	30,000,000 BZPX	Prevents whale capture of the emission share.
Lock duration	90 – 2,555 days	Mandatory; continuous; capped at time-to-emission-end.
Max LTV	50%	Borrow ceiling against effective stake; prevents looping.
Liquidation threshold	95%	Position becomes liquidatable at this LTV.
Liquidation bonus	5%	Surplus seized above debt, paid to the liquidator.
Reserve factor	3%	Protocol's cut of interest.
Boost (linear, quad)	750, 250 bps	Quadratic lock-boost coefficients, base 10000.
Flash-loan guard	10 blocks	Minimum hold before withdraw; defeats same-block attacks.
Conservation dust	10^{10} wei	Per-transaction conservation tolerance (10^{-8} BZPX).

Two of these are *immutable* in the strongest sense — the BZPX token address and the treasury address are set at construction and can never change — and one design choice deserves emphasis: the 30-million-per-wallet cap. Emission is shared in proportion to boosted stake; without a cap, a single large holder could capture a disproportionate slice and crowd out the small stakers the protocol most wants to retain. The cap makes emission progressive in the only sense that matters on-chain: it bounds the share any one address can claim.

19.2 The Master Conservation Identity

Most lending and staking contracts treat solvency as something to *monitor* — a dashboard, an off-chain alert, a health metric an operator watches. BlazePhoenix treats solvency as something to *enforce*. The entire contract is subordinate to one equation, the Master Conservation Identity, which relates the real token balance the contract holds to the sum of every claim against it:

THE MASTER CONSERVATION IDENTITY

$$\underbrace{\text{balance}(\text{this}) + \text{badDebt}}_{\text{what is held + written off}} = \underbrace{(S - D) + \mathcal{R}_{\text{rew}} + \mathcal{R}_{\text{prot}} + (\Pi_{\text{dist}} - \Pi_{\text{paid}})}_{\text{owed}(\cdot)} \tag{22}$$

where S is total staked, D total debt, $\mathcal{R}_{\text{rew}}$ the unspent emission reserve, $\mathcal{R}_{\text{prot}}$ the protocol reserve, and $\Pi_{\text{dist}} - \Pi_{\text{paid}}$ the rewards distributed into accumulators but not yet withdrawn. The right-hand side is the function `owed()`; the contract is **solvent** exactly when its real balance meets or exceeds it. This is not an assertion in a comment — it is a public, free, on-chain view (`solvency()`, `isSolvent()`, `collateralRatio()`) that anyone can read, and it is enforced on every transaction that moves value by a modifier we call the **Conservation Guard**.

THE CONSERVATION GUARD — SOLVENCY AS A REVERT CONDITION

Every value-moving entry-point — deposit, borrow, repay, withdraw, claim, liquidate — is wrapped in a `conserves` modifier. It snapshots the real balance and `owed()` before the body runs, and after the body requires that the *change* in real balance equals the *change* in `owed()`, to within a dust tolerance of 10^{-8} BZPX. A transaction that would leave the ledger claiming even slightly more than the contract holds does not merely get flagged — it reverts. An insolvent state is therefore not "monitored"; it is **unreachable**.

The guard is deliberately framed on the *per-transaction delta*, not the absolute balance. Were it absolute, any pre-existing rounding dust — or a donation, or a fee-on-transfer quirk — could wedge the whole contract by making the absolute identity fail, a denial-of-service by invariant. By checking only that each transaction conserves value relative to the state it inherited, historical drift cancels and only a transaction that *itself* leaks value is rejected. The result is a contract that is not just believed to be solvent but is structurally incapable of becoming insolvent through its own operation.

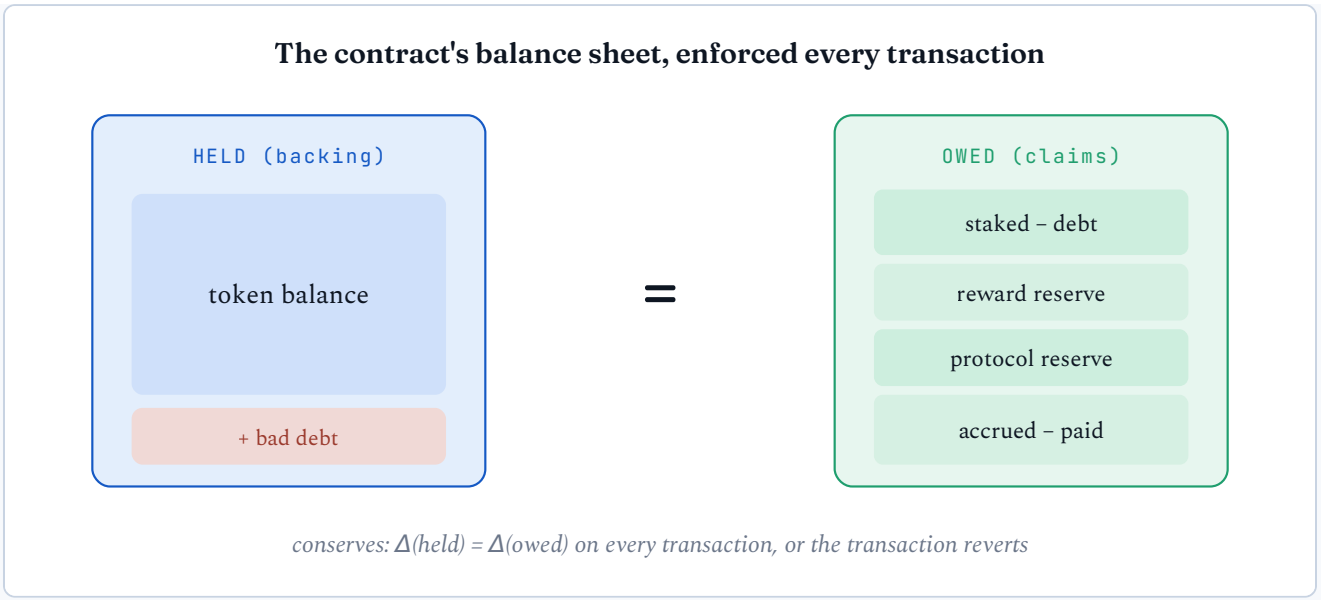

FIG. 18 *The Master Conservation Identity. What the contract holds equals what it owes, at all times. The* `conserves` *modifier proves the per-transaction change of each side is equal before committing — so the contract cannot move from a solvent state to an insolvent one. Solvency is a property of the code, readable by anyone for free.*

19.3 No oracle, by construction

Lending protocols are most often broken not at the math but at the oracle: a manipulated price feed makes sound collateral look worthless or worthless collateral look sound, and the liquidation logic — however correct — executes on a lie. BlazePhoenix removes the oracle entirely, not by trusting a better one but by making one unnecessary. The collateral asset and the borrowed asset are the *same token*: a staker locks BZPX and borrows BZPX against it. A position's loan-to-value is therefore a pure ratio of two BZPX quantities — debt over stake — that needs no external price to evaluate.

> **AN ABSENT ATTACK SURFACE**
>
> There is no price feed to manipulate, no oracle to delay, no TWAP window to game, because there is no price in the system at all. The most exploited primitive in decentralized lending is simply not present. Liquidation triggers on $\text{debt} \cdot 100 \geq \text{stake} \cdot 95$ — arithmetic over two amounts of one token — and that comparison is as manipulation-proof as the token balances themselves.

19.4 The quadratic boost curve

Locking stake for longer should earn more — but the schedule of "more" is a design choice with consequences. A linear boost under-rewards commitment; a steep exponential one concentrates power in the longest lockers. BlazePhoenix uses a gentle quadratic curve in the lock duration d, measured in days:

BOOST CURVE (BASIS POINTS, BASE 10000)

$$\beta(d) \;=\; 10000 \;+\; 750 \cdot \frac{d}{365} \;+\; 250 \cdot \left(\frac{d}{365}\right)^2 \qquad (90 \le d \le 2555) \tag{23}$$

The quadratic term is deliberately small relative to the linear one, so the curve rewards commitment without runaway concentration. The resulting multipliers range from about $1.02\times$ at the 90-day minimum to $2.75\times$ at the seven-year maximum, passing through $1.10\times$ at one year, $1.25\times$ at two, and $2.00\times$ at five.

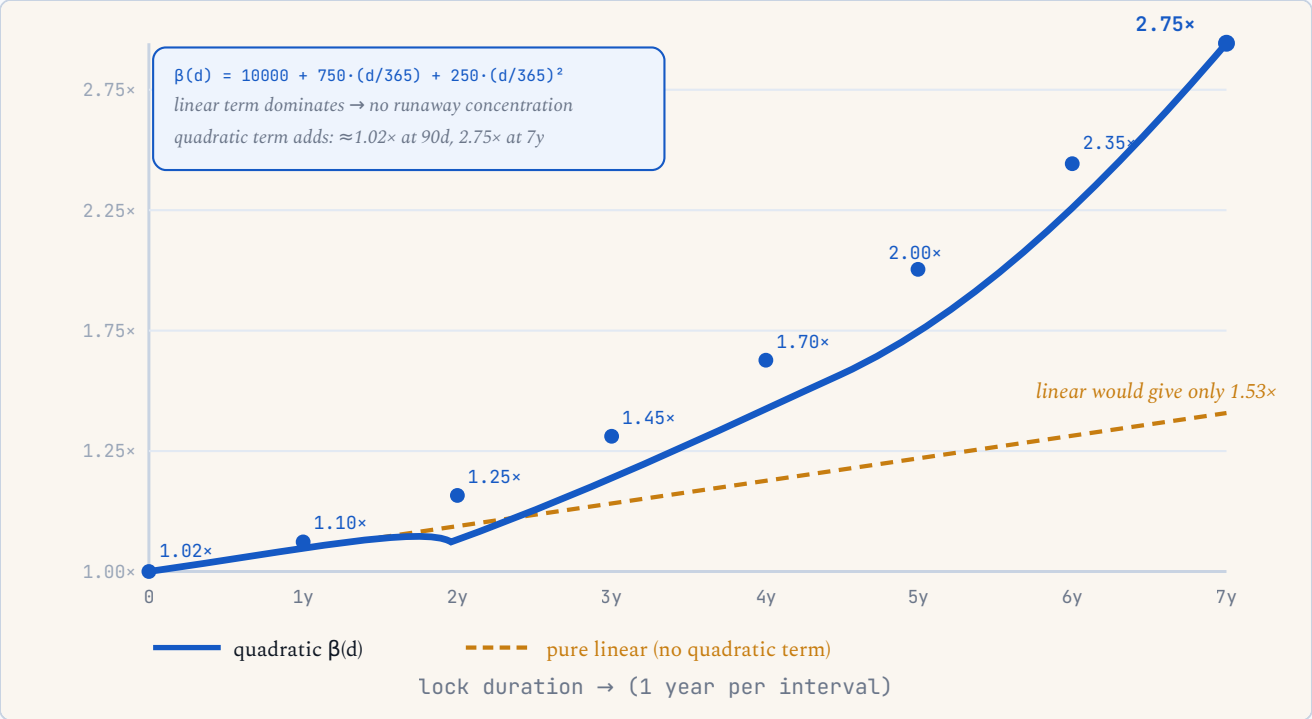

FIG. 19 *The continuous quadratic boost $\beta(d) = 10000 + 750 \cdot (d/365) + 250 \cdot (d/365)^2$. Each year of commitment earns progressively more than the year before. A purely linear schedule would top out at 1.53× at seven years; the mild quadratic term lifts the ceiling to 2.75× without allowing runaway concentration — a maximal locker earns under triple a minimal locker, while a pure linear boost at the same rate would give less incentive to commit beyond year three.*

Crucially, the boost applies to a staker's *own* stake, multiplicatively. It does not dilute other stakers' principal; it re-weights the claimant's share of the emission accumulator. A small staker who locks for seven years earns the same $2.75\times$ multiplier on their position as a whale does on theirs — the boost is scale-free, favouring neither large nor small.

19.5 Mandatory locked staking

BlazePhoenix has no liquid stake. Every deposit commits its funds for a chosen duration between 90 days and roughly seven years, and the duration is a *decreasing countdown* capped at the time remaining to the emission end — there is no point locking past the last block that pays a reward. A top-up may only *extend* the lock, never shorten it, so a position's commitment is monotone. This is a deliberate inversion of the usual incentive: rather than taxing early exit with a penalty, the protocol simply does not offer one.

> NO EARLY-EXIT, NO PENALTY — AND NO NEED FOR EITHER
>
> A staker withdraws principal only when two conditions both hold: the lock has expired, and the position is fully debt-free. There is no early-withdrawal fee, because there is no early withdrawal; there is no penalty schedule to mistune, because the lock is simply honoured. Borrowing remains available throughout — a staker can draw liquidity against locked collateral at any time — but the stake itself is released only when its commitment is complete and its debt is zero. Commitment is enforced by time, not punished by fees.

The mandatory lock is also what makes the dual-yield accounting of §19.6 clean: because a borrower's collateral cannot vanish mid-loan, the relationship between a position's stake, its debt, and its boosted contribution stays well-defined across the entire life of the loan, and the global accumulators always divide by a denominator that reflects committed capital rather than capital that might flee at the first sign of stress.

19.6 Dual yield — two accumulators, one base

The Engine pays two distinct yields from two distinct sources, and a staker earns one or both depending on whether they borrow. This is the most subtle part of the design, and the part that lets a single contract serve both lenders and pure savers without cross-subsidy. The accounting follows the MasterChef pattern: a global accumulator tracks reward-per-share, each user stores a debt marking the accumulator value at their last settlement, and pending yield is the difference scaled by the user's boosted share.

The first accumulator distributes the protocol emission across the total *boosted-effective* stake — every staker's unencumbered, boost-weighted capital:

REWARD ACCUMULATOR (EMISSION → BOOSTED-EFFECTIVE STAKE)

$$\texttt{accReward} \mathrel{+}= \frac{\dot{R} \cdot \Delta t \cdot \mathrm{WAD}}{S_{\mathrm{eff}}}, \qquad \mathrm{pending}_u = \frac{e_u \cdot \texttt{accReward}}{\mathrm{WAD}} - \texttt{rewardDebt}_u \tag{24}$$

where S_{eff} is the total boosted-effective stake and e_u the user's boosted-effective contribution. The emission is **deterministic**: if $S_{\mathrm{eff}} = 0$ over some interval — no eligible stake — the accumulator does not advance and the unspent emission simply stays in reserve, its clock advancing so a latecomer can never arrive and capture a backlog of rewards no one was owed. The second accumulator is fed not by emission but by the *interest borrowers pay*, distributed across the total boosted-pure stake (the boost-weighted stake of users who carry no debt):

PURE-YIELD ACCUMULATOR (BORROWER INTEREST → PURE STAKERS)

$$\texttt{accPureYield} \mathrel{+}= \frac{I_{\mathrm{net}} \cdot \mathrm{WAD}}{S_{\mathrm{pure}}}, \qquad I_{\mathrm{net}} = I \cdot \left(1 - \tfrac{300}{10^4}\right) \tag{25}$$

with I the interest accrued, the 3% reserve factor skimmed to the protocol reserve, and S_{pure} the total boosted-pure stake. The two accumulators never mix. A borrower's effective stake (stake minus debt) earns emission; a pure staker's full stake earns both the emission and the interest stream. Each party is paid for exactly what they contribute: the pure staker is, in effect, the lender of last resort — they supply the stake borrowers draw against, and they are paid the interest that borrowing generates — while a borrower forgoes the pure-yield stream on the portion of stake they have encumbered but keeps emission on the rest.

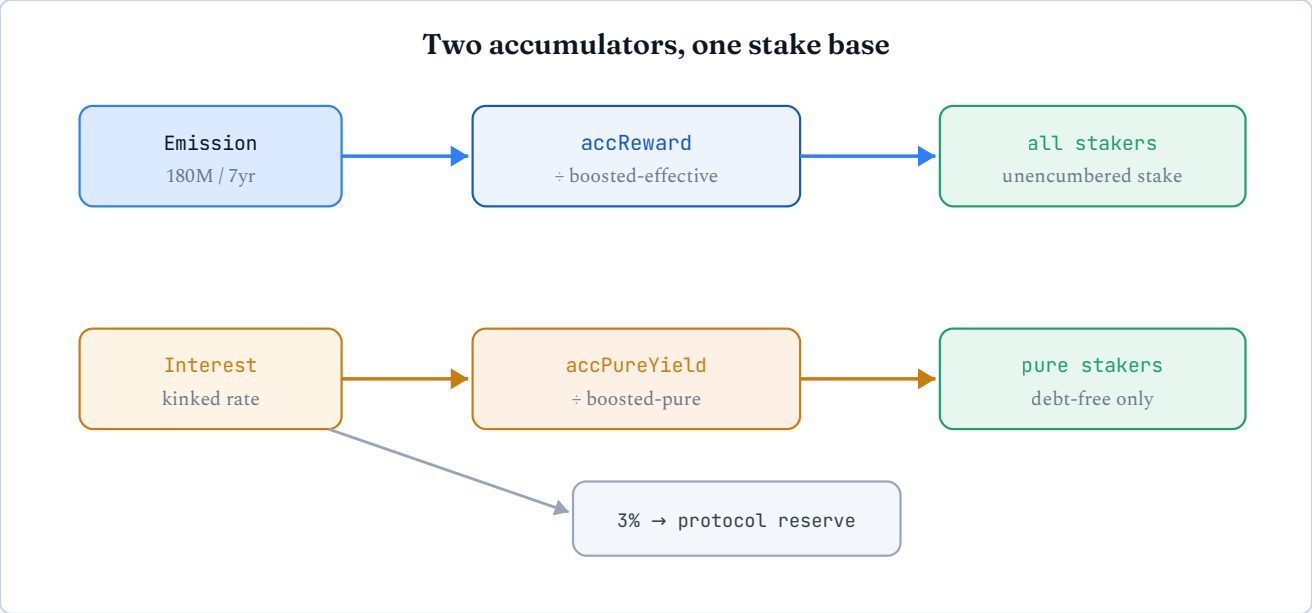

FIG. 20 *The Dual-Accumulator Doctrine. Emission flows through* `accReward` *to every staker's boosted-effective stake; borrower interest flows through* `accPureYield` *to debt-free stakers, net of a 3% reserve cut. The two populations are paid from disjoint sources, so neither subsidises the other and the books always balance.*

19.7 Effective stake and the boosted totals

Every yield computation rests on a single derived quantity: a user's boosted contribution to each pool. For a user with stake s, debt b and lock duration d, the boost $\beta = \beta(d)$ gives:

BOOSTED-EFFECTIVE AND BOOSTED-PURE CONTRIBUTIONS

$$e_u = \frac{\max(0,\ s - b) \cdot \beta}{10^4}, \qquad p_u = \begin{cases} \dfrac{s \cdot \beta}{10^4}, & b = 0 \ \wedge \ s > 0 \\ 0, & \text{otherwise} \end{cases} \tag{26}$$

The asymmetry is the dual-yield rule made precise. Boosted-effective stake nets out debt — a borrower earns emission only on the capital they have not encumbered. Boosted-pure stake is non-zero only for users with no debt at all — the moment a staker borrows, they leave the pure-yield population entirely and join the emission-only population on their encumbered portion. The contract enforces this through a **single writer**: one internal routine, and only that routine, mutates the global boosted totals, re-synchronising both contributions and adjusting the denominators atomically whenever stake, debt or lock changes. No path — not deposit, not liquidation, not even the emergency hatch — is permitted to desync the totals the accumulators divide by. This single-writer discipline is what guarantees that the per-share arithmetic of equations (24)–(25) is always dividing by a denominator that reflects reality.

19.8 The credit facility

A staker may borrow BZPX against their stake up to a 50% loan-to-value ratio on their effective stake. The borrow cap is exact:

MAXIMUM BORROW

$$b_{\max} = \frac{(s - b) \cdot 50}{100} \tag{27}$$

Borrowing accrues interest continuously at a rate set by a kinked utilisation curve — the same jump-rate model proven across the major lending protocols, parametrised here for a single-asset market. Let utilisation $U = D/S$. Below a kink at $U^\star = 80\%$ the rate rises gently from a 1% floor to 5%; above it, steeply, to choke off the last of the liquidity and protect the withdrawal buffer of stakers:

KINKED INTEREST-RATE MODEL (ANNUAL BASIS POINTS)

$$r(U) = \begin{cases} 100 + 500\,U, & U \le 0.8 \\ 500 + 72500\,(U - 0.8), & U > 0.8 \end{cases} \tag{28}$$

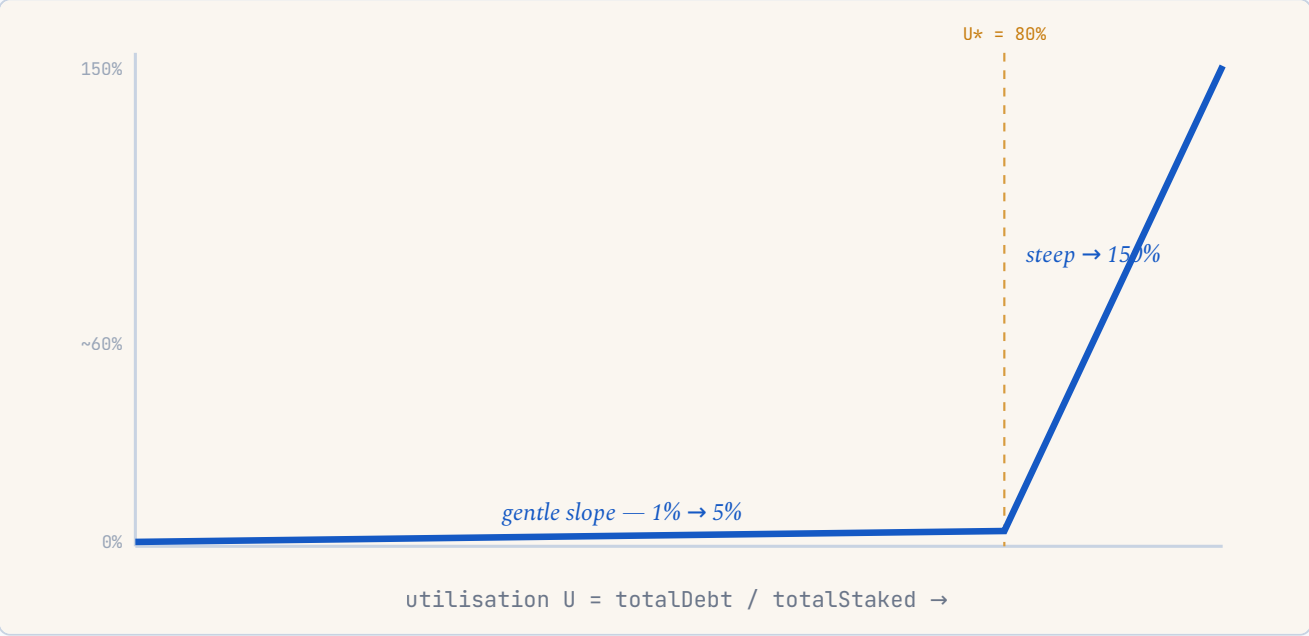

FIG. 21 *The kinked interest-rate curve. Below 80% utilisation the rate climbs gently from a 1% floor to 5% at the kink; past it the rate climbs steeply, reaching punitive levels as utilisation approaches 100%. The steep tail makes the last units of borrowable stake expensive, preserving a withdrawal buffer for stakers.*

Interest is accrued per position on a continuous-by-elapsed-time basis. For a position with debt b held for Δt seconds at rate r:

PER-POSITION INTEREST ACCRUAL

$$I = \frac{b \cdot r}{10^4} \cdot \frac{\Delta t}{\text{seconds per year}} \tag{29}$$

The accrued interest is deducted from the borrower's own stake — the position self-amortises — with 3% routed to the protocol reserve and 97% paid into the pure-yield accumulator. Interest, in other words, is the precise transfer from borrowers to pure stakers that equation (25) distributes. The system has no external interest source; it is a closed loop, and every unit of yield a pure staker receives is a unit of interest a borrower paid. (Should interest ever exceed a borrower's remaining stake — possible only under extreme, sustained neglect — the excess is booked transparently rather than silently socialised, exactly as bad debt is in §19.9.)

19.9 Liquidation and bad-debt coverage

A position becomes liquidatable when its loan-to-value reaches the 95% threshold — that is, when debt has grown, through accrued interest, to within 5% of the stake backing it:

LIQUIDATION CONDITION

$$\text{liquidatable} \iff b \cdot 100 \geq s \cdot 95 \tag{30}$$

On liquidation, the protocol seizes the debt plus a 5% bonus, capped at the available stake. The seized amount flows through a waterfall: the debt is cleared, the bonus is paid to the liquidator as the incentive that funds the gas of keeping the book clean, and any stake beyond the seizure is returned to the user — who becomes, by virtue of now carrying no debt, a pure staker:

LIQUIDATION SEIZURE

$$\text{seized} = \min\left(b + \tfrac{b \cdot 5}{100},\ s\right), \qquad \text{leftover} = \max(s - \text{seized},\ 0) \tag{31}$$

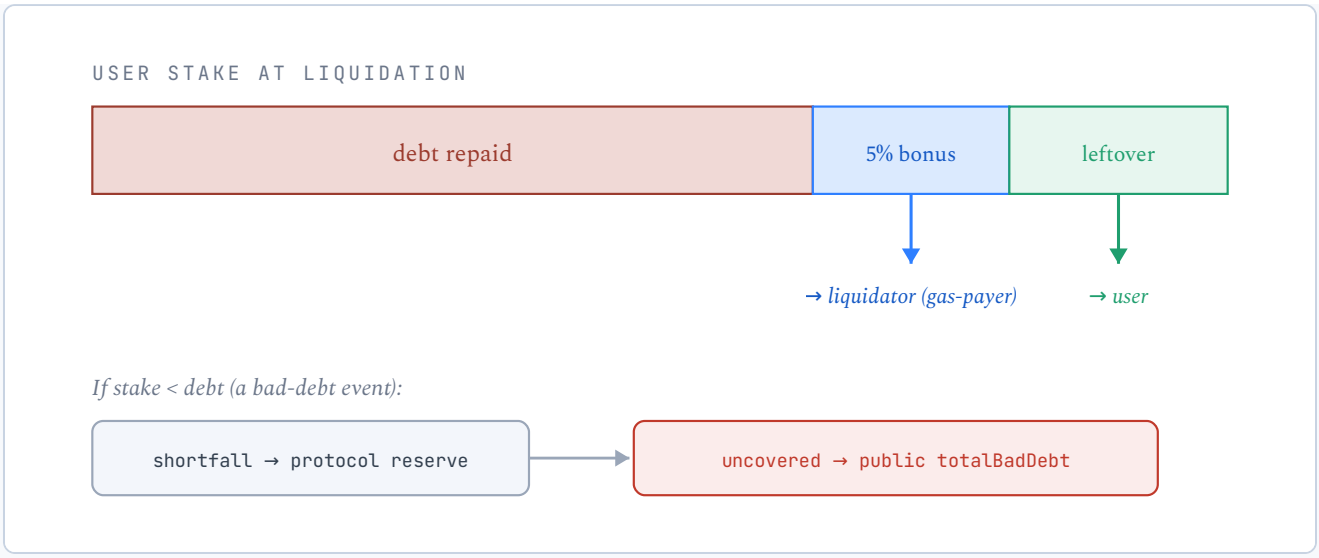

FIG. 22 *The liquidation waterfall. The seized stake repays debt, releases a 5% bonus to the liquidator, and returns any leftover to the user. If stake has fallen below debt, the shortfall is covered first from the protocol reserve, with any uncovered remainder booked transparently to a public* `totalBadDebt` *counter — no loss is socialised without first appearing on the protocol's own books, in plain sight.*

The bad-debt path is where many lending designs hide their fragility. BlazePhoenix makes it explicit. When a position's stake has fallen below its debt — possible only under extreme, sustained neglect given the autonomous maintenance of §20 — the shortfall is first absorbed by the protocol reserve, and whatever the reserve cannot cover is recorded in a public `totalBadDebt` counter and emitted as an event. Crucially, bad debt enters the Master Conservation Identity of §19.2 directly: it sits on the held side of the ledger as an acknowledged write-off, so the contract's solvency view always reflects it honestly. There is no socialisation of loss across stakers without it first appearing on the protocol's own books.

20 Autonomous Maintenance & the Solvency Breaker

Liquidations must be timely, but scanning every position on every block is prohibitively expensive, and the industry's usual answer — a fleet of keeper bots watching the chain and firing liquidations off-chain — reintroduces precisely the operational dependency this protocol refuses everywhere else. If the bots stop, the book rots. BlazePhoenix removes them. The book cleans *itself*, from the organic flow of ordinary use.

20.1 The self-cleaning book

Every ordinary user action — deposit, borrow, repay, withdraw, claim, lock, or a no-op `poke` — drives an internal routine that performs a small, gas-bounded sweep over the borrower set. The sweep walks a rotating window of positions from a persistent cursor: it liquidates any that have gone underwater, paying the seizure surplus to whoever supplied the gas, and it *pokes* the rest — accruing their interest, settling their yield, processing any expired lock, and re-synchronising their boosted contribution — so the global accumulator denominators never drift. The cursor advances each call, so over a sequence of transactions the entire borrower set is swept, and a busy market maintains itself continuously while a quiet one falls behind only as far as its own inactivity warrants.

A permissionless `liquidate(user)` entry-point remains for anyone who wishes to act deliberately — a keeper, an arbitrageur, a concerned staker — but it is an *option*, not a *dependency*. The protocol does not require a single keeper to exist; it merely rewards anyone who chooses to be one. The borrower registry is itself public and paginated, so a would-be liquidator can enumerate at-risk positions on-chain without any indexer.

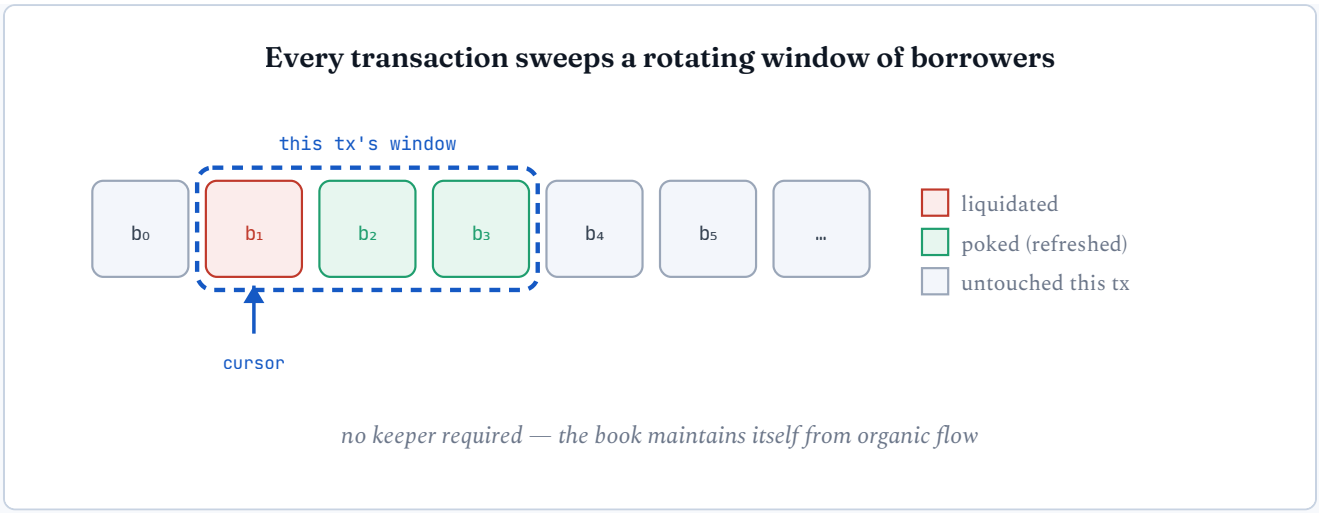

FIG. 23 *Autonomous maintenance. Each user transaction sweeps a gas-bounded window of the borrower set from a rotating cursor, liquidating the underwater and refreshing the rest. The surplus from any liquidation pays the gas of whoever triggered it, so maintenance is funded by the very flow that performs it.*

20.2 Health-adaptive cadence

The sweep window is not fixed; it self-sizes with backlog pressure. The number of positions examined per transaction grows with the borrower count and with the time elapsed since the last sweep, capped to bound worst-case gas:

MAINTENANCE BUDGET (POSITIONS PER TRANSACTION)

$$c \;=\; \min\!\Big(\,10,\; n,\;\; 1 + \Big\lfloor \tfrac{n}{50} \Big\rfloor + \Big\lfloor \tfrac{t - t_{\text{last}}}{900} \Big\rfloor \,\Big) \tag{32}$$

with n the borrower count and $t - t_{\text{last}}$ the seconds since the last maintenance run (one extra position scanned per 50 borrowers and per 15 minutes of gap, capped at ten). The budget is a pure function of on-chain state — there is no administrator knob — so maintenance effort rises automatically exactly when the backlog warrants it and never exceeds a fixed gas ceiling. The protocol can also tell any borrower, at any moment, exactly how long they have before liquidation at the current rate, because the estimate is closed-form:

DAYS TO LIQUIDATION (CLOSED FORM)

$$T_{\text{liq}} \;=\; \frac{s - s^{\dagger}}{\text{daily interest burn}}, \qquad s^{\dagger} = \left\lceil \tfrac{b \cdot 100}{95} \right\rceil \tag{33}$$

where $s^{\dagger}$ is the stake level at which the position becomes liquidatable. A transparency most lending markets cannot offer follows immediately: the runway is the buffer of stake above the liquidation point divided by the daily interest burn. One safety property deserves emphasis: maintenance is **disabled while the protocol is paused or in emergency**. A borrower who cannot act to defend their position can never be liquidated by someone else's transaction — the lifeboat and the liquidator are never armed at the same time.

20.3 The permissionless breaker

The protocol carries two halt paths, and the more important of the two trusts no one's judgement. A **guardian** may declare an emergency at discretion — the conventional pause. But anyone at all may call `tripBreaker`, and it succeeds *only* when the blockchain itself proves the contract is insolvent: when the real balance plus dust has fallen below what the contract owes. The condition is objective and un-spoofable — it is the Master Conservation Identity of §19.2 read in the breach direction — so the breaker cannot be tripped maliciously on a healthy contract, and cannot be *prevented* from tripping on a genuinely broken one.

> INSOLVENCY IS A FACT, NOT AN OPINION
>
> A discretionary pause depends on someone noticing, deciding, and acting. The permissionless breaker depends on arithmetic: if `balance + dust < owed`, any address can halt the contract, and no privileged party can stop them. If the breach was transient — a momentary accounting artefact rather than a true loss — the admin can reverse the breaker, but only after the chain confirms the hard invariant holds again. Trust is removed from the halt decision in both directions: you cannot falsely trip it, and you cannot falsely suppress it.

20.4 The pull-only emergency hatch

When an emergency is active, the contract opens a minimal-logic `emergencyWithdraw` by which any staker recovers their own net equity — stake minus debt — through the simplest possible path: read the position, net it, transfer, zero the record. It deliberately avoids the reward settlement, interest accrual and boost re-synchronisation that constitute the contract's complex surface, so if a vulnerability lives anywhere in that surface, the escape route does not touch it. The lifeboat is built from different timber than the ship. To preserve the accounting that other stakers still depend on, the hatch funnels its single balance change through the same single-writer that governs the

boosted totals, so even an emergency exit cannot desync the global denominators — and, uniquely, the hatch is *not* wrapped in the conservation guard, so it remains callable even in a hard-breached state. The escape works precisely when everything else has stopped.

> ZERO BACKDOOR
>
> There is no administrative path that sweeps staker principal — not after a grace period, not ever. The only value an administrator can move is the protocol's own reserve, and even that flows exclusively to an *immutable* treasury address fixed at construction: no arbitrary destination, no discretion over where it goes. A compromised admin key cannot redirect a single staker's BZPX, because the function that would do so does not exist. An emergency hatch that let an administrator sweep would be indistinguishable from a backdoor; this one cannot become one.

Every one of these properties is independently checkable on-chain. A public `auditInvariants` view returns a bitmask flagging any violation of five structural invariants at once — conservation, solvency, the emission cap, accumulator monotonicity, and boost bounds — so the contract's health is not a claim made in this document but a value anyone can read from the chain, for free, at any block.

21 The Verification Surface

A protocol that custodies value should not ask to be believed; it should publish the means to be checked. Most staking and lending contracts expose balances and little else — to know whether the system is actually solvent, you must trust the team's dashboard or reconstruct the state from an indexer. The BlazePhoenix Staking Engine turns its own integrity into a set of public, free, on-chain views, so that any party — a staker, an auditor, a competitor, a sceptic — can confirm in a single `eth_call` that the contract holds what it claims to owe.

21.1 Solvency, as a public read

The Master Conservation Identity of §19.2 is not hidden inside the implementation; it is surfaced. Five views report it directly. `owed()` returns the right-hand side of equation (22) — the total of all claims. `backing()` returns the real token balance the contract holds. `isSolvent()` returns the boolean comparison. `collateralRatio()` returns the ratio of the two, scaled so that exactly 10^{18} means precisely solvent and any greater value means an over-collateralised surplus. And `solvency()` returns the full report — backing, owed, ratio, total bad debt, and the solvency flag — in one struct. None of these requires an argument, an oracle, or an off-chain service; they are pure reads over the contract's own storage.

> COLLATERAL RATIO, READ LITERALLY
>
> A single integer tells the whole story. `collateralRatio() == 1e18` means the contract holds exactly what it owes; `> 1e18` means it holds a surplus (the reserve and rounding buffer); `< 1e18` would mean a deficit — a state the `conserves` guard makes unreachable through ordinary operation, and which, were it ever reached through some external pathology, any address could act on through the permissionless breaker of §20.3. Solvency is therefore not a quarterly attestation; it is a number on the chain, true at every block.

21.2 The five-invariant audit

Beyond the conservation identity, the contract checks itself against five structural invariants and packs the result into a single bitmask returned by `auditInvariants()`. A return of zero means every invariant holds; any set bit localises a violation.

TABLE 20 — THE AUDITINVARIANTS BITMASK

Bit	Invariant	Meaning
0	Conservation	The held side equals the owed side within dust — equation (22) holds now.
1	Solvency	Real balance meets or exceeds total claims; the contract can honour every withdrawal.
2	Emission cap	Cumulative distributed rewards never exceed the fixed 180M budget.
3	Monotonicity	Both reward accumulators only ever increase — no path rewinds a settled share.
4	Boost bounded	Each tracked boosted total stays within stake × the maximum boost — no inflated share.

21.3 Position and discovery views

The same transparency extends to individual positions and to the borrower set as a whole. Any staker can read their `pendingRewards` and `pendingPureYield` before claiming, their `healthFactor`, and a closed-form `daysToLiquidation` (§20.2) telling them exactly how long their position survives at the current rate. And because autonomous maintenance and permissionless liquidation both need to enumerate borrowers, the borrower registry is itself public and paginated — `activeBorrowerCount`, `borrowerAt`, `getBorrowers`, `isTrackedBorrower`, and the current `maintenanceBudget` — so a would-be liquidator finds at-risk positions directly on the chain, with no subgraph and no indexer in the loop.

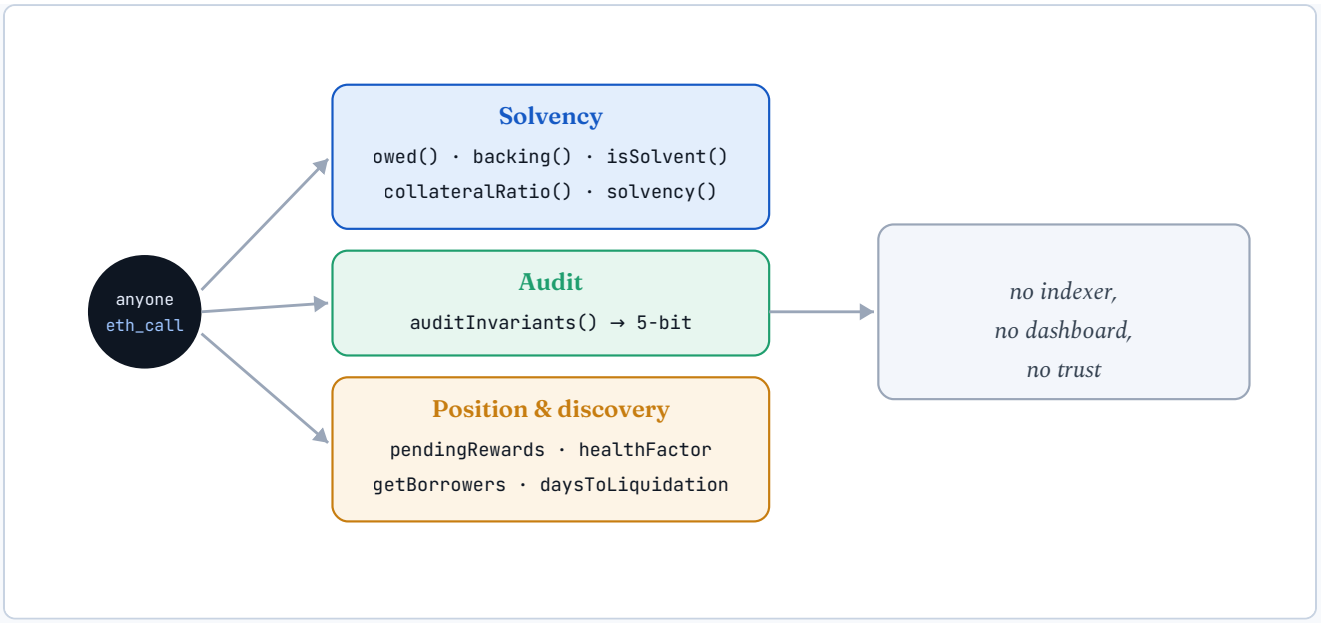

FIG. 26 *The verification surface. Solvency, the five-invariant audit, and every position and borrower fact are public on-chain reads. A third party confirms the protocol's integrity in a single call, with no indexer and no privileged access — the protocol does not ask to be trusted, it asks to be verified.*

22 Economic Security

The Engine's safety rests on arithmetic that an adversary cannot bend, not on assumptions an adversary can break. Because there is no oracle and no off-chain dependency, the threat model is unusually small, and each threat that remains is closed by a structural property rather than a watchful operator. We walk the principal ones.

22.1 The loan-to-value buffer

A position may be opened at a loan-to-value of at most 50% and becomes liquidatable only at 95%. Because collateral and debt are the same token, that 45-percentage-point gap is a buffer measured in real terms — it is not a price cushion that a market move can erase, but a fixed quantity of accrued interest that must accumulate before a healthy position becomes unhealthy. A borrower at the maximum 50% must let interest roughly double their debt relative to stake before liquidation, and the self-amortising interest of §19.8 gives them clear, closed-form warning of exactly when that will happen.

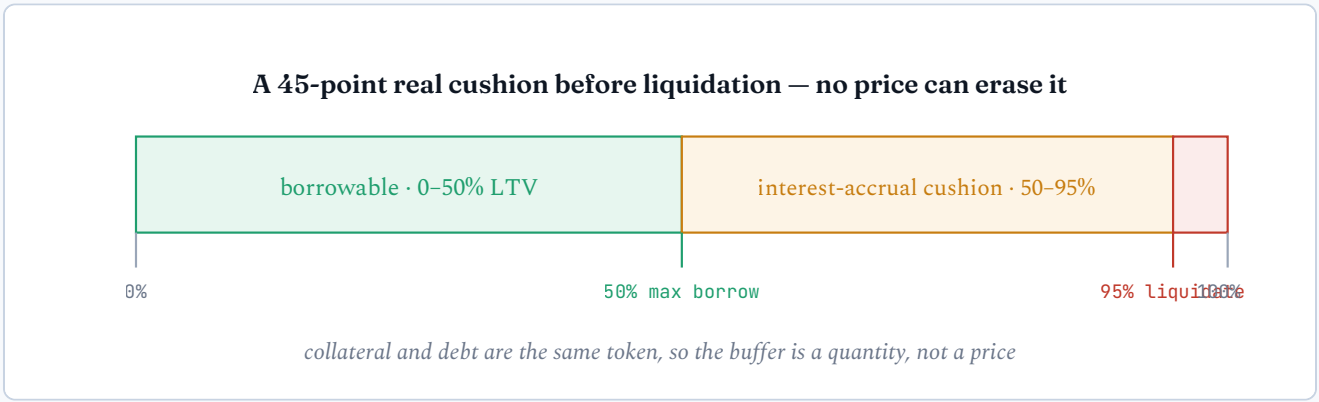

FIG. 27 *The loan-to-value buffer. Borrowing is capped at 50% LTV; liquidation triggers at 95%. The intervening 45 points are a fixed real-terms cushion of accrued interest — immune to the price shocks that break oracle-dependent lenders, because there is no price in the system.*

22.2 Why borrowing cannot drain the contract

A borrow sends BZPX out of the contract, but it increases the borrower's recorded debt by the same amount, so the `conserves` guard proves the held side and the owed side move together and the transaction is solvency-neutral. The borrower cannot draw more than half their effective stake, the position stays over-collateralised by construction, and the only way to extract value beyond

one's own equity is to let interest carry the position past 95% — at which point liquidation seizes the collateral, repays the debt, and pays a surplus to the liquidator. There is no sequence of borrows, repays, or stake changes that leaves the contract holding less than it owes, because every such step is checked against the conservation identity before it commits.

22.3 The remaining threats, and what closes them

TABLE 21 — STAKING ENGINE THREAT MODEL

Threat	What closes it
Oracle manipulation	No oracle exists; collateral and debt are the same token (§19.3). The dominant lending-exploit class is structurally absent.
Keeper failure / liquidation lag	Autonomous maintenance liquidates from organic flow (§20.1); no keeper is required to exist, and a permissionless one is rewarded if it does.
Administrative rug	No path sweeps staker principal; reserve withdrawals flow only to an immutable treasury (§20.4). The function to redirect funds does not exist.
Flash-loan / same-block attack	A 10-block minimum hold before withdrawal, and a per-transaction conservation guard that no single-block manipulation can satisfy while leaking value.
Interest insolvency	Interest is a closed loop — every unit paid to a saver is a unit a borrower paid — and the conservation guard rejects any accrual that would unbalance the ledger.
Backlog capture	Deterministic emission advances its clock over empty intervals (§19.6), so a latecomer cannot claim rewards accrued while no eligible stake existed.
Bad debt	Covered first from reserve, remainder booked to a public counter that enters the solvency identity directly (§19.9). No silent socialisation.
Whale capture of emission	A 30M-per-wallet stake cap bounds any single address's share of the boosted-effective base.
Reentrancy	The conservation guard plus checks-effects-interactions ordering plus the single-writer boost discipline; transient claims cannot re-enter to double-settle.

The list is short by design. An invariant-driven contract over a single asset, with no oracle and no keeper, simply has fewer places to fail — and the places that remain are closed by properties the EVM enforces on every transaction, not by vigilance that can lapse.

23 The Lifecycle of a Position

The mechanics of §19–§22 are best understood by following a single position through its life. We trace one, with concrete numbers, to show how deposit, lock, boost, dual yield, borrowing, self-amortising interest and finally withdrawal compose into one coherent record — and how the contract keeps its books balanced at every step.

23.1 From deposit to boosted stake

A staker deposits 1,000,000 BZPX and chooses a five-year lock. The boost curve of equation (23) gives $\beta(1825) = 2.00\times$, so their boosted contribution is 2,000,000. Because the deposit carries no debt, the position is, from the first block, a *pure staker*: it contributes 2,000,000 to the boosted-effective base (and thus earns emission on its full stake) and 2,000,000 to the boosted-pure base (and thus earns the borrower-interest stream as well). The lock is recorded as a countdown; a later top-up could only extend it. The flash-loan guard means the stake cannot be withdrawn for at least ten blocks regardless — but with a five-year lock, that constraint is moot.

23.2 The dual yield in numbers

Suppose the protocol's total boosted-effective stake is 100,000,000. The annual emission is $180\text{M}/7 \approx 25.71\text{M}$ BZPX, shared in proportion to boosted-effective contribution. Our staker holds $2\text{M}/100\text{M} = 2\%$ of the base, so they earn roughly 514,000 BZPX per year in emission on a 1,000,000 stake — an emission yield near 51% on principal at this utilisation of the base, entirely a function of how much boosted stake competes for the fixed budget. On top of that, as a pure staker they receive their share of the interest stream that borrowers pay, distributed across the boosted-pure base. A borrower with the same stake and lock who draws 400,000 BZPX (a 40% LTV, within the 50% ceiling) instead contributes 1,200,000 to the boosted-effective base — earning emission only on their unencumbered 600,000 of stake — and zero to the boosted-pure base, forgoing the interest stream entirely while they carry debt. Each party is paid for exactly what they expose.

TWO YIELDS, ONE STAKE, NO CROSS-SUBSIDY

The pure staker earns emission *and* interest; the borrower earns emission only, on the portion of stake they have not encumbered. Neither subsidises the other: emission comes from the fixed budget and interest comes from borrowers, and the two accumulators of §19.6 are filled from disjoint sources. A staker chooses their position on this spectrum — pure saver, or borrower trading interest income for liquidity — and the contract prices both correctly from the same record.

23.3 The borrowing arc and self-amortisation

Follow the borrower. The 400,000 debt accrues interest at the kinked rate of equation (28); that interest is not billed separately but *deducted from the borrower's own stake*, so the position self-amortises — stake falls, debt is serviced, and the loan-to-value climbs over time. The closed-form `daysToLiquidation` tells the borrower, at any moment, exactly how many days remain until the LTV reaches 95% at the prevailing rate. They may repay at any time to reset the trajectory, top up stake to push the liquidation point further out, or do nothing and let autonomous maintenance liquidate the position once it crosses the threshold — at which point the debt is cleared, a 5% surplus pays whoever's transaction performed the liquidation, and any leftover stake returns to them as a now-debt-free pure-staking position.

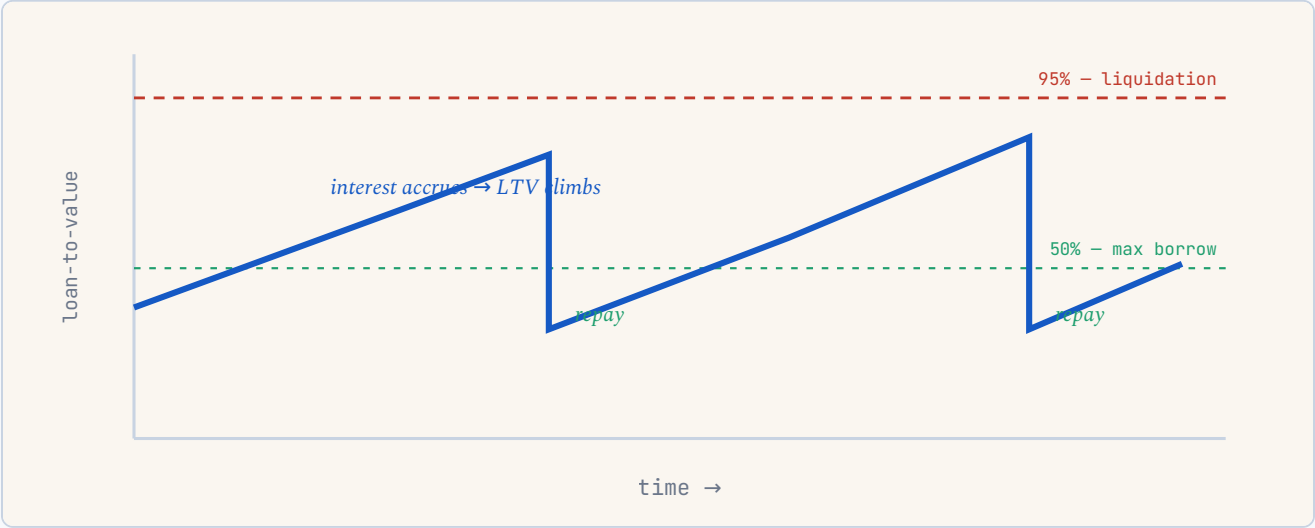

FIG. 28 *A borrower's loan-to-value trajectory. Self-amortising interest lifts the LTV over time toward the 95% liquidation line; a repayment resets it. The borrower always knows the runway, because* `daysToLiquidation` *is closed-form.*

23.4 Closing the position

A staker withdraws principal only when the lock has expired *and* the debt is zero — the two conditions of §19.5. There is no fee for waiting and no way to skip it. When both hold, the withdrawal returns the stake, the position's boosted contributions are removed from the global totals through the single writer, and — if the stake goes fully to zero — the record is deleted. The figure below shows the full state machine the position traverses.

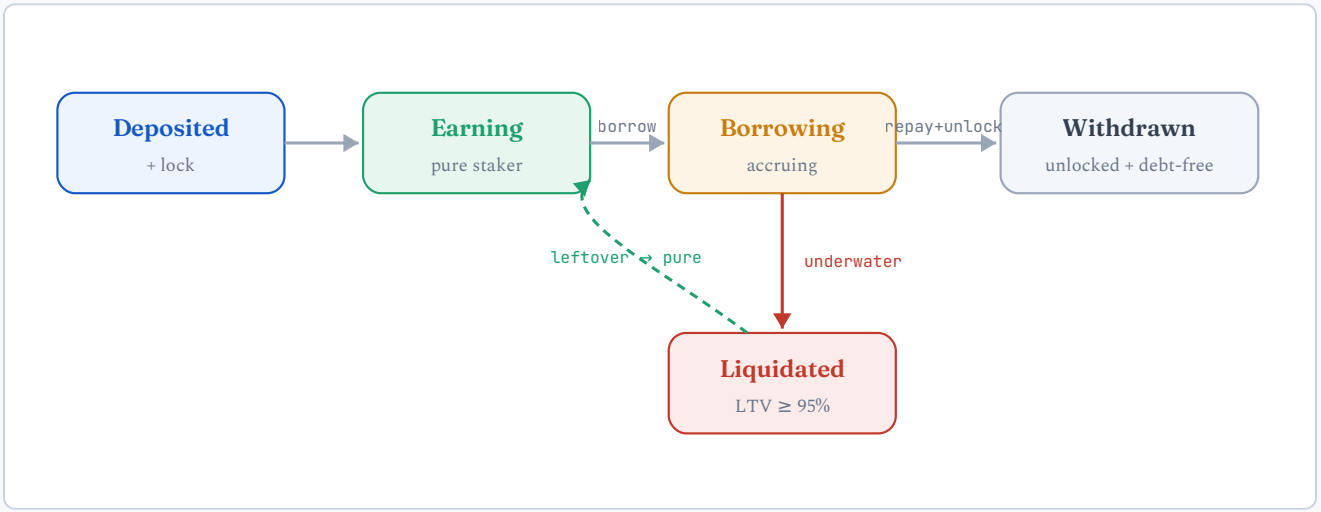

FIG. 29 *The position state machine. A deposit locks and earns; the staker may borrow and later repay, or be liquidated if the LTV crosses 95% (leftover stake returns as a pure position). Withdrawal is reachable only once the lock has expired and the debt is zero.*

24 Stateless · Serverless · Ungovernable — the Staking Edition

The aggregator earned the three words of §16; the Staking Engine earns them again, on harder ground. A yield-and-lending venue is exactly the kind of system that the industry routinely props up with oracles, keeper fleets, upgradeable proxies and custodial admin keys. BlazePhoenix builds one with none of them, and the absence of each is a property you can verify rather than a claim you must accept.

Stateless.

The boost curve, the kinked rate, the dual accumulators and the conservation identity are pure arithmetic, and the heavy lifting — full-precision multiply-divide, the raw token transfers that the conservation guard measures — lives in a pure `MathLib` library with no storage of its own. The contract's economic behaviour is a function of its state in the mathematical sense: the same balances always yield the same yields, the same solvency answer, the same liquidation decision, with nothing learned from off-chain and nothing retuned by an owner.

Serverless.

There is no keeper, because autonomous maintenance liquidates and refreshes from organic flow. There is no price oracle, because collateral and debt are the same token. There is no indexer, because the borrower set and every solvency figure are public on-chain reads. Nothing in the critical path depends on a process someone must keep running: the book maintains itself, the solvency proves itself, and the at-risk positions enumerate themselves.

Ungovernable.

Administrative authority is bounded to the point of near-irrelevance and is itself renounce-able. The admin can fund the one-time emission and withdraw accrued protocol reserve — and that reserve flows only to an immutable treasury address, never to an arbitrary destination, never including a staker's principal. A guardian can pause; anyone can trip the breaker on proven insolvency; and the role keys can be renounced outright. There is no upgrade path, no proxy, and no function anywhere that moves a staker's locked BZPX to an address the staker did not authorise.

TABLE 22 — TWO ARCHITECTURES FOR ON-CHAIN CREDIT

Concern	Web 2.5 lending	BlazePhoenix Staking
Collateral pricing	External price oracle (manipulable, laggy)	None — collateral and debt are one token
Liquidation	Off-chain keeper bots	Autonomous, from organic flow; permissionless option
Solvency	Operator dashboard / attestation	Public on-chain identity, enforced per transaction
Upgrade surface	Admin-controlled proxy	No proxy, no upgrade path
Custody of principal	Often an admin-reachable path	No path exists; reserve only to immutable treasury
Risk parameters	Off-chain governance, mutable	Immutable constants in the contract
Halt authority	Privileged multisig	Guardian pause + permissionless insolvency breaker

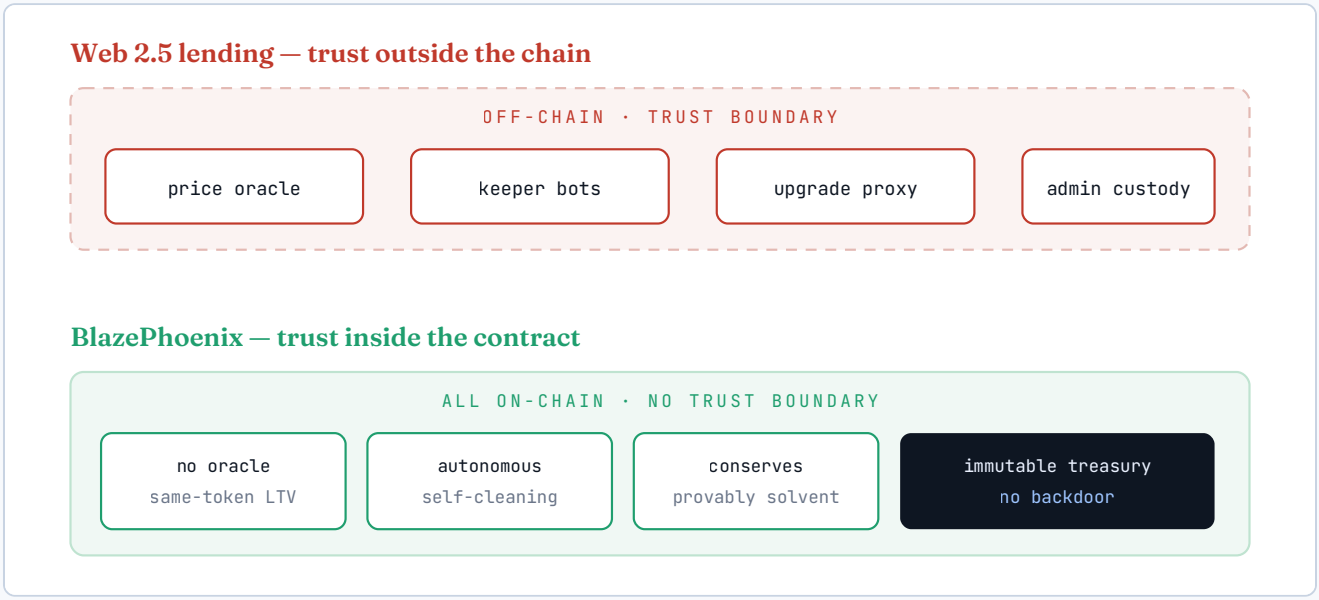

FIG. 30 *Two architectures for on-chain credit. The Web 2.5 lender depends on an oracle, a keeper fleet, an upgrade key and a custodial path — each a trust boundary the user cannot verify. BlazePhoenix has none: no price, no keeper, a per-transaction solvency proof, and an immutable treasury with no path to staker principal.*

"A vault that proves its own solvency on every transaction, liquidates without a keeper, prices without an oracle, and cannot be drained by the key that deploys it. This is not a lending protocol with

decentralisation bolted on — it is a lending protocol that has nothing left to centralise.”

THE BLAZEPHOENIX THESIS, APPLIED TO CAPITAL

25 Tokenomics

A token earns its supply schedule by what that schedule defends against. BZPX is fixed at one billion units, allocated to maximise tradable liquidity and long-term alignment while keeping insider supply small, vested, and transparent. The largest share by far is committed to the market itself; the team's allocation is the smallest of the major buckets and the only one locked behind a multi-month vesting cliff.

The native token is **Blaze Phoenix** (**BZPX**), with a fixed total supply of **1,000,000,000** tokens. There is no inflation beyond this cap: the staking emission of §19 is drawn from a pre-allocated slice of the fixed supply, not minted anew. Once the emission slice is exhausted over its seven-year schedule, no further BZPX can ever be created — a property the fundEmission entry-point enforces by accepting the budget only once and capping it at 180M.

25.1 Allocation

FIG. 31 *The BZPX allocation across six buckets. Market liquidity dominates at 60%, the staking emission reserves 18%, and the combined insider allocation (team, partners) is 17.3% — of which the team's 11.3% is the only vested tranche.*

The supply divides into six buckets, each with a distinct economic purpose:

TABLE 23 — BZPX ALLOCATION

Bucket	Tokens	Share	Purpose & schedule
Market liquidity	600,000,000	60.0%	Deep, tradable liquidity across venues from launch; the float that lets the aggregator do its work.
Staking emission	180,000,000	18.0%	The fixed reward budget of §19, released linearly over seven years to stakers.
Team & Operations	113,000,000	11.3%	Vested over 30 months, released monthly — the only locked insider tranche.
Partners	60,000,000	6.0%	Strategic integrations, venue partnerships, ecosystem grants.
Airdrop	40,000,000	4.0%	Distribution to early users and the community.
Initial marketing	7,000,000	0.7%	Launch awareness and onboarding.
Total	**1,000,000,000**	**100%**	Fixed supply; no inflation beyond this cap.

25.2 Vesting and emission schedule

Two of the six buckets release over time rather than at once; the rest are liquid at genesis. The team allocation follows a linear monthly unlock over 30 months, and the staking emission follows the constant rate of equation (21) over seven years.

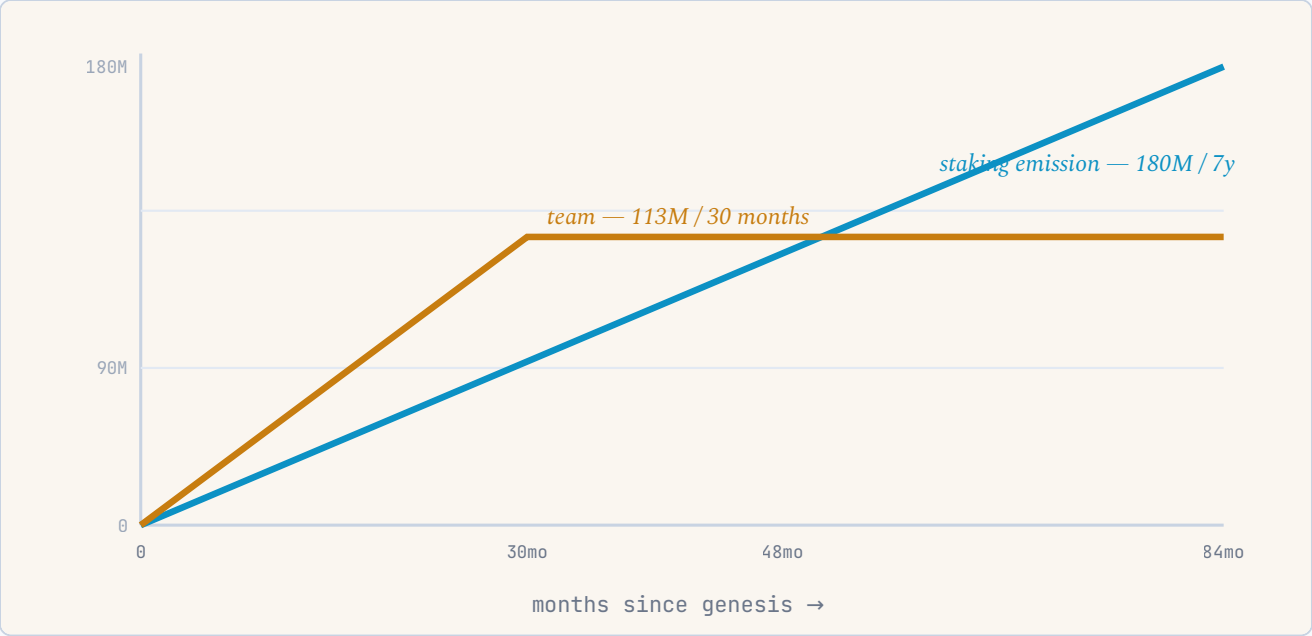

FIG. 32 *The two scheduled buckets release on deliberately different timescales — the team's over thirty months, the stakers' over seven years — so that long-term holders, not insiders, hold the longest-tail claim on new supply.*

SUPPLY DISCIPLINE

60% to the market, 18% to stakers over seven years, 11.3% to the team locked across 30 months. The structurally small and vested insider share, against a dominant liquidity float and a long emission tail, is the token-level expression of the same principle that governs the code: align incentives, bound the privileged, and make every commitment legible in advance. Because the staking emission is carved from the fixed supply and the team tranche vests linearly, the fully-diluted supply is known and fixed from genesis — there is no mechanism, administrative or otherwise, to mint beyond one billion BZPX. The token's scarcity is a property of the contract, not a policy that can be revised.

26 Conclusion

We set out to build a complete on-chain protocol on a single discipline — compute, do not trust — and to carry that discipline from routing other people's liquidity all the way to custodying the protocol's own capital. The two engines that result share more than a token; they share a method.

The aggregator showed that liquidity fragmentation is a problem of language, not of mathematics: beneath the proliferation of venues lies a small family of invariants, and a protocol that expresses each once — the Eightfold Dispatcher, the Deterministic Derivation, the Iron-Law floor, the Vitality Field — can aggregate the whole landscape without per-venue special-casing, pricing every concentrated venue by the venue's own bytecode so the quote cannot diverge from the fill. The Staking Engine showed that solvency itself can be a property of the code rather than a claim about it: a single conservation identity, enforced on every transaction, makes an insolvent state unreachable; a single-asset design removes the oracle that breaks most lenders; and autonomous maintenance keeps the book clean with no keeper at all. Both engines are stateless at their core, serverless in their operation, and ungovernable in the dimensions where governance is a weapon — and in both, the only privileged powers that remain are ones that cannot touch user funds, and even those can be surrendered.

> THE THESIS, RESTATED
>
> Six shapes, not sixty. One Core, imported everywhere, redefined nowhere. A registry that learns liquidity by trading it, and a vault that proves its own solvency by arithmetic. A number quoted off-chain that equals the number realised on-chain; a balance sheet that balances on every transaction or the transaction reverts. No oracle, no keeper, no indexer, no backdoor. This is what it means for a protocol to live on a chain rather than merely settle on one — and it is what Web 3.0 was always supposed to be.

What remains is operational, not mathematical: independent external audit of the fund-moving and accounting paths, the migration of administrative control to a multi-signature scheme and then through the one-way door, and the gradual extension of each engine — new venue shapes in the Core, new collateral behaviour in the vault — added only when their mathematics is proven and verified against ground truth, never guessed. The protocol does not ask to be trusted. It asks to be verified, and it makes verification cheap, public, and permanent.

“The shapes are few, not sixty; the invariants are few, and they are known. Implement each once, correctly, prove it, and the rest is dispatch.”

ANONYMOUS MITRA · JUNE 2026

A Appendix — Derivations

This appendix records the derivations behind the staking formulae of §19–§23, for the reader who wishes to check the contract's arithmetic against first principles. Throughout, basis points use the base 10^4 and the fixed-point scale is $\mathrm{WAD} = 10^{18}$.

A.1 The boost multipliers

Evaluating $\beta(d) = 10000 + 750\,(d/365) + 250\,(d/365)^2$ at integer-year locks, with each term floored as the contract computes it, gives the schedule below. The quadratic term contributes nothing at one year and grows to nearly half the total only at the seven-year extreme — the source of the curve's mild convexity.

TABLE 24 — BOOST MULTIPLIER BY LOCK DURATION

Days	Years	β (bps)	Multiplier
90	0.25	10199	1.02×
365	1	11000	1.10×
730	2	12500	1.25×
1095	3	14500	1.45×
1460	4	17000	1.70×
1825	5	20000	2.00×
2190	6	23500	2.35×
2555	7	27500	2.75×

A.2 Continuity of the kinked rate

The rate model of equation (28) is continuous at the kink $U^\star = 0.8$. The lower branch evaluates to $100 + 500 \cdot 0.8 = 500$ bps; the upper branch evaluates to $500 + 72500 \cdot (0.8 - 0.8) = 500$ bps. The two agree, so the curve has no jump. The slope ratio across the kink is $72500/500 = 145$: borrowing the last fifth of available liquidity is priced 145 times more steeply than the first four-fifths, which is what preserves a withdrawal buffer for stakers as utilisation approaches one.

A.3 The dual accumulator is O(1)

For the emission stream, the global accumulator grows by $\Delta\texttt{accReward} = \dot{R}\,\Delta t\,\mathrm{WAD}/S_{\text{eff}}$ over an interval. A user who last settled when the accumulator read $\texttt{accReward}_0$ stored $\texttt{rewardDebt}_u = e_u\,\texttt{accReward}_0/\mathrm{WAD}$. Their pending reward is therefore

PENDING REWARD AS A PER-SHARE DIFFERENCE

$$\text{pending}_u = \frac{e_u \cdot \texttt{accReward}}{\mathrm{WAD}} - \texttt{rewardDebt}_u = \frac{e_u}{\mathrm{WAD}}\left(\texttt{accReward} - \texttt{accReward}_0\right) \tag{A.1}$$

— the user's boosted share of the per-share growth since they last settled, computed in constant time with no iteration over users or over time. Settlement simply resets $\texttt{rewardDebt}_u$ to the current gross. The pure-yield accumulator works identically over S_{pure}, which is why a single contract can pay two streams to thousands of positions at O(1) cost per interaction.

A.4 Per-operation conservation

The `conserves` guard requires $\Delta\,\text{backing} = \Delta\,\text{owed}$ for every value-moving operation, where $\text{owed} = (S - D) + \mathcal{R}_{\text{rew}} + \mathcal{R}_{\text{prot}} + (\Pi_{\text{dist}} - \Pi_{\text{paid}})$. The three elementary cases are immediate:

CONSERVATION, THREE ELEMENTARY OPERATIONS

$$\begin{aligned} \textbf{deposit}(x): &\quad \Delta\text{backing} = +x, &\quad \Delta(S-D) = +x \\ \textbf{borrow}(x): &\quad \Delta\text{backing} = -x, &\quad \Delta(S-D) = -x \\ \textbf{claim}(c): &\quad \Delta\text{backing} = -c, &\quad \Delta(\Pi_{\text{dist}} - \Pi_{\text{paid}}) = -c \end{aligned} \tag{A.2}$$

Interest accrual moves value entirely *within* the owed side: a borrower's stake falls by I (reducing $S - D$ by I), the reserve rises by $0.03\,I$, and the pure-yield claims rise by $0.97\,I$, summing to zero — and the backing does not move, so the guard passes. Liquidation is the same construction: the seized stake repays debt (both S and D fall together, neutral), the 5% bonus leaves the contract and is matched one-for-one by the extra fall in stake over debt, and any shortfall is booked as bad debt on the held side. No reachable operation breaks the equality, which is precisely why insolvency is unreachable.

A.5 Days to liquidation

A position is liquidatable when $b \cdot 100 \geq s \cdot 95$, i.e. when stake falls to $s^{\dagger} = \lceil 100b/95 \rceil$. Interest burns stake at the annual amount $b\,r/10^4$, so the daily burn is $b\,r/(10^4 \cdot 365)$ and the runway is the buffer over the liquidation level divided by that burn:

LIQUIDATION RUNWAY

$$T_{\text{liq}} = \frac{s - \lceil 100b/95 \rceil}{b\,r/(10^4 \cdot 365)} \ \text{days} \tag{A.3}$$

A.6 Pure-staker yield

A position's emission yield, per unit of staked principal, is its boost times the annual emission divided by the boosted-effective base: $\text{APR}_{\text{emit}} \approx \beta \cdot (\dot{R} \cdot \text{year})/S_{\text{eff}}$. The staked principal cancels, so the yield depends only on the lock boost and on how much boosted stake competes for the fixed budget — the more capital staked, the lower the per-unit emission, as one expects of a fixed reward pool. A pure staker additionally earns the interest stream $\text{APR}_{\text{pure}} \approx \beta \cdot 0.97\,(\text{annual interest})/S_{\text{pure}}$; a borrower earns only the emission term, on their unencumbered stake.

B Appendix — Parameters & Notation

For reference, the staking notation used throughout §19–§23 and Appendix A is collected below, alongside the immutable constants that constitute the entire tunable surface of the Engine.

TABLE 25 — STAKING NOTATION

Symbol	Meaning
$\beta(d)$	Lock-boost multiplier (bps, base 10000) for a lock of d days; $1.02\times$–$2.75\times$.
s, b	A position's stake and debt, both in BZPX.
e_u, p_u	A user's boosted-effective and boosted-pure contributions, equation (26).
S_{eff}, S_{pure}	Global totals of boosted-effective and boosted-pure stake (the accumulator denominators).
U	Utilisation, total debt over total staked.
$r(U)$	Annual borrow rate (bps) from the kinked model, equation (28).
accReward	Emission reward-per-share accumulator; pays the boosted-effective base.
accPureYield	Interest reward-per-share accumulator; pays the boosted-pure base.
$\dot{R}$	Linear emission rate, equation (21); 180M BZPX over seven years.
owed()	Total claims against the contract — the right-hand side of the conservation identity.
backing()	The contract's real BZPX balance — the held side of the identity.
$s^{\dagger}$	The stake level at which a position becomes liquidatable, $\lceil 100b/95 \rceil$.

TABLE 26 — IMMUTABLE PARAMETERS

Constant	Value
Total emission budget	180,000,000 BZPX
Emission period	7 years
Max stake per wallet	30,000,000 BZPX
Lock duration	90 – 2,555 days
Max LTV / liquidation threshold	50% / 95%
Liquidation bonus / reserve factor	5% / 3%
Boost coefficients (linear, quadratic)	750, 250 bps
Interest floor / kink / kink-rate	1% / 80% / 5%
Flash-loan guard	10 blocks
Conservation dust	10^{10} wei
Maintenance scan cap	10 positions / tx

References & Glossary

[1] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008.

[2] H. Adams, N. Zinsmeister, D. Robinson. *Uniswap v2 Core.* 2020.

[3] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, D. Robinson. *Uniswap v3 Core.* 2021.

[4] R. Bloemen. *Math for full-precision multiply-divide (mulDiv) in 256-bit fixed-width arithmetic.* 2019.

[5] M. Egorov. *StableSwap — Efficient Mechanism for Stablecoin Liquidity.* Curve Finance, 2019.

[6] A. Cronje et al. *Solidly: the stable-and-volatile invariant.* 2022.

[7] Aerodrome Finance. *Aerodrome: a MetaDEX for Base.* Protocol documentation, 2023.

[8] F. Vogelsteller, V. Buterin. *EIP-1167: Minimal Proxy Contract.* Ethereum Improvement Proposals, 2018.

[9] Ethereum Foundation. *EIP-1014: Skinny CREATE2.* Ethereum Improvement Proposals, 2018.

[10] H. Adams et al. *Uniswap v4 Core: the singleton PoolManager and hooks.* 2024.

[11] Uniswap Labs. *Permit2: a unified token-approval contract.* 2022.

[12] V. Buterin et al. *EIP-7702: Set EOA account code.* Ethereum Improvement Proposals, 2024.

[13] Ethereum Foundation. *ERC-7201: Namespaced Storage Layout.* 2023.

[14] R. Leshner, G. Hayes. *Compound: The Money Market Protocol.* 2019. (Jump-rate utilisation model.)

[15] SushiSwap. *MasterChef: staking reward accounting.* 2020. (Reward-per-share accumulator pattern.)

Glossary of operators

TABLE 27 — PROTOCOL OPERATORS

Symbol	Meaning
$\mathscr{U}$(route)	Aggregator route utility — the meta-equation value maximised by the Solver.
Ω	The Eightfold Dispatcher; one closed-form output per AMM shape.
$\mathscr{D}$	Deterministic CREATE2 address derivation.
Φ	Iron-Law feasibility floor; a pure indicator on net output.
Ψ	Liquidity-vitality field; the trust weight of a pool.
Ξ	Anti-scam projector; zeroes any route touching a hostile or unlisted hook.
β(d)	Staking lock-boost multiplier; the bridge between the two engines' notation.

This whitepaper is published under the name Anonymous Mitra.

BlazePhoenix Protocol · Complete Technical Whitepaper · Version 1.0.0 · June 2026 · License BUSL-1.1