

BLAZEPHOENIX PROTOCOL · TECHNICAL WHITEPAPER

Blaze*Phoenix*

An on-chain liquidity aggregator whose quote and execution are one computation

The quote is the trade.

TECHNICAL WHITEPAPER — VERSION 2.3.1

Written by ***Mitra***

Version	2.3.1 · corrects 2.3 (2026-09-24)
Date	2026-11-16
Author	Mitra
License	CC BY-NC-ND 4.0 (text) · BUSL-1.1, source-available (code)
Web	https://blazephoenix.xyz
Source	https://blazephoenix.xyz/whitepaper.md

Contents

Ten lines. No marketing. 5

Abstract. 5

How to read this paper. 6

The Innovations. 6

I-1 · Quote ≡ Execution. 6

I-2 · The on-chain solver, free to call. 7

I-3 · The Meta-Equation. 7

I-4 · The Eightfold Dispatcher Ω. 7

I-5 · Deterministic Derivation $\mathcal{D}$. 7

I-6 · Uniswap V4, entirely in-frame. 8

I-7 · The Solidly stable solver. 8

I-8 · The Monoslot. 8

I-9 · The Self-Healing Registry and the Vitality Field Ψ. 8

I-10 · The Capital-Anchored Split. 9

I-11 · The Iron-Law floor Φ. 9

I-12 · Measurement at every seam. 9

I-13 · The One-Callback Doctrine. 9

I-14 · The Surplus Rule and immunity by exhaustion. 10

I-15 · The byte–gas exchange rate ρ^*. 10

I-16 · The anti-scam projector Ξ. 10

I-17 · The One-Way Door. 10

I-18 · Metrological Design and the published denominator. 10

The scoreboard. 11

§1 The problem: a quote is a promise someone else computed. 11

§1.1 The fragmentation is real, and shallower than it looks. 12

§1.2 The off-chain escape hatch. 12

§2 The answer and the discipline. 13

§2.1 Invariant-driven design. 14

§2.2 Why this discipline fits this medium. 15

§2.3 Metrological Design: one morphism, measured coordinates, monotone inputs. 15

§3 The Meta-Equation and the dependency budget. 16

§3.1 The dependency budget. 18

§3.2 Three edges. 19

§4 The Eightfold Dispatcher Ω. 19

§4.1 Six kinds, four families, one attribute table. 20

§4.2 The closed forms. 21

§4.3 Two fee units, carried natively. 22

§4.4 One dispatcher, two surfaces. 22

§5 Uniswap V4 end to end. 22

§5.1 Locating a pool that has no address: derive, then prove. 23

§5.2 Reading a pool that answers no calls. 25

§5.3 Settling with a singleton ... 25
§5.4 The fee that exists only at execution time ... 25
§5.5 Hooks: screened by arithmetic, never by interrogation ... 26
§5.6 What the integration is worth ... 26
§6 The Solidly solver ... 26
§7 Deterministic Derivation $\mathcal{D}$ ... 29
§8 The Monoslot ... 31
§9 The Self-Healing Registry and the Vitality Field Ψ ... 33
§9.1 Learning by trading, and the pair-proof ... 33
§9.2 The score ... 34
§9.3 Eviction ... 35
§9.4 Discovery, and its gas knob ... 35
§9.5 The configuration surface ... 35
§9.6 The structural boundary ... 36
§10 Routing and the Split Allocator ... 36
§10.1 Route shapes and budgets ... 37
§10.2 The funnel ... 37
§10.3 The believability band ... 38
§10.4 The allocator and its three guards ... 38
§11 The execution layer and the Iron-Law floor Φ ... 39
§11.1 Four doors, one executor ... 39
§11.2 Measurement, from the first instruction ... 40
§11.3 Three floors under every leg ... 41
§11.4 One callback, one unlock ... 41
§11.5 The Iron-Law floor Φ, in full ... 42
§11.6 Settlement, and how to read one ... 44
§11.7 The Router's refusals ... 45
§12 Quote surfaces, and how a quote ages ... 45
§12.1 The Quoter's surfaces ... 46
§12.2 How a quote ages ... 47
§12.3 What a quote costs through the ABI ... 48
§13 Fee policy: the two regimes and the Surplus Rule ... 49
§13.1 Where the fee is charged: two regimes, both named ... 49
§13.2 The commitment has one producer, and the contract counts ... 50
§13.3 The Surplus Rule ... 50
§13.4 The fee, measured from outside the Router ... 51
§14 Security model ... 53
§14.1 Attacks and the checks that refuse them ... 54
§14.2 Hostile hooks: three layers, cheapest first ... 57
§14.3 Bounded, not eliminated: staleness, ordering, and the sandwich curve ... 59
§14.4 Hostile venues ... 60
§14.5 The hostile front end ... 60
§14.6 What a hostile key can and cannot do ... 61
§14.7 The One-Way Door ... 61

§14.8 What this model asks of you ... 62
§15 The verification apparatus, measured ... 62
§15.1 The suite in numbers ... 62
§15.2 The evidence chain, and the curated mutation guard ... 63
§15.3 Mutants aimed at the invariants ... 63
§15.4 Regime covering arrays ... 65
§15.5 The hostile-venue matrix ... 66
§15.6 Canonical oracles ... 66
§15.7 Metamorphic relations ... 67
§15.8 N-version: the same source under other code generation ... 68
§15.9 Projection distance ... 68
§15.10 Evidence independence, the shared-quantity register, and the artefact ... 69
§15.11 What none of this establishes ... 69
§16 Architecture and sizes ... 70
§16.1 Five contracts: what each holds, what each may call ... 70
§16.2 The life of one swap ... 72
§16.3 Sizes against EIP-170 ... 73
§16.4 The attack surface, counted ... 74
§17 Measured performance ... 76
§17.1 Cross-chain identity ... 76
§17.2 Preview → delivery fidelity across four chains ... 76
§17.3 The factory census ... 77
§17.4 Gas ... 77
§17.5 The executor against the field ... 78
§17.6 The byte–gas exchange rate ρ^* ... 79
§18 The DEX, stripped to the bone ... 79
§19 Deployment status ... 80
§20 Token — BZPX ... 81
§21 Research programme and security record ... 81
§22 What you must trust, and what you need not ... 83
§23 Related work ... 84
§24 Conclusion ... 85
Revision history ... 86
Version 2.3.1 (2026-11-16) ... 86
Version 2.3 (2026-09-24) ... 86
Errata to 2.2 (2026-09-07) ... 86
Errata to 2.2 (2026-09-24) ... 87
Appendix A · The primitive equations ... 88
Appendix B · The invariants ... 89
Appendix C · Reproduction commands ... 91
Glossary ... 93
References ... 93

> ***Version note — 2.3.1 (2026-11-16).*** *This version corrects three statements of version 2.3 (2026-09-24) and adds this note, a dated status line and a revision-history entry; nothing else changed, and the text still describes* `Blaze-Phoenix-Dex` *at* `main` `8916545`*. (1) The code licence's Change Date is 2030-06-01, as the repository's LICENSE file states at* `8916545` *and at* `969bcc66b7d2eddd8dbece45dd5cc13deba77164`*; version 2.3 printed 2030-07-01. (2) The staking line of the allocation table in §20 now states the emission design of the staking contract at commit* `c0366cf30095d39741596df29d1d5be590765955` *(*`src/BlazePhoenixStaking.sol`*, lines 188-216): two independent deployments, each capped at 90,000,000 BZPX over 8 years, a linear base of 60,000,000 plus a bonus of 30,000,000 that halves every two years. The 180,000,000 total is unchanged. (3) The text licence of this version is CC BY-NC-ND 4.0; version 2.3 keeps the licence it was published under, CC BY 4.0. The code is described as source-available. Status on 2026-11-16: V2 is a pre-launch engineering preview with no mainnet contract address, the deployed generation is still V1, and the BZPX token has not launched.*

> *The price you are shown is computed by the public code that executes your trade, in the frame that executes it. Nobody — the authors included — can change the fee or the floors: they are compiled in.*

Ten lines. No marketing.

1. **No server picks your route.** The transaction that executes it derives it, from public pool state.
2. **No oracle prices your trade.** The pools do — read in the frame that settles it.
3. **The quote and the floor are one evaluator.** There is no seam for the price to drift through.
4. **Your minimum is mandatory.** The Router refuses a swap without one.
5. **A pool's claims are free to forge; its measured output is not.** Routing runs on the second.
6. **Fee, split and floors are compiled in.** No setter exists. Nobody — the authors included — can move them.
7. **No proxy, no upgrade path, no** `selfdestruct`**, no balance at rest.** There is nothing to steal from the Router between trades.
8. **A hostile pool costs you price, then reverts.** It never delivers below your floor.
9. **Every number here is printed beside its denominator** — and beside how it would be gamed.
10. **If every server we run vanished tonight, the contracts would quote, route and settle tomorrow.**

Everything below is the proof, with the line of code that enforces each claim and the limit that bounds it.

Abstract

BlazePhoenix-Dex is an on-chain liquidity aggregator built on one claim: a quote and its execution are the same computation. The route is derived inside the transaction that executes it, from public pool state, by a stateless library any reader can re-run; the number that binds at settlement is re-derived by the same dis-

patcher in the executing frame. There is no routing service, no price oracle, no keeper and no upgrade path. This paper specifies the five contracts — Core, Hub, Solver, Router, Quoter — and the operators of the Meta-Equation: the Eightfold Dispatcher Ω, the Deterministic Derivation $\mathcal{D}$, the Iron-Law floor Φ, the Vitality Field Ψ and the anti-scam projector Ξ. Behind them sits one discipline, Metrological Design: measure every coordinate that can be measured, and make the rest monotone so a false input hurts only its supplier. Every guarantee is stated with its enforcement site and its limit. The verification apparatus is printed with its denominators: 1,609 declared tests across 236 files, 268 of 268 curated mutants killed, 14 of 15 mutants aimed at 43 stateful invariants noticed, covering arrays of strength 2 (63 rows, 258 pairs) and 3 (168 rows, 1,636 triples), 14 + 5 metamorphic relations, three optimiser builds in agreement, and a 240-sample study of how a quote ages. Eighteen named innovations carry the design, from the Meta-Equation to immunity by exhaustion; each is catalogued with its mathematics before the specification begins. The generation live on Base, Ethereum, Optimism and Arbitrum is V1; this paper specifies V2, exercised end to end on forks of live liquidity on every chain it serves.

How to read this paper

Every mechanism gets a plain sentence before it gets a precise one, so the paper reads at two depths: the plain sentences and the figures give the architecture; the precise sentences, the tables and the appendices let you check it. Every guarantee is one sentence — the claim, the site that enforces it, the limit that bounds it. Six worked examples carry the arithmetic through by hand: a constant-product quote (§4), a split across two pools (§10), the Iron-Law floor on a two-hop route (§11), a Uniswap V4 pool identifier derived from its key (§5), the two fee regimes on one route (§13), and the reading of a settlement (§11). Numbers are printed beside the denominator they were measured against, and the command that reproduces them is in Appendix C. The repository is canonical: where this text and the source disagree, the source wins.

The Innovations

BlazePhoenix-Dex is not an aggregator with a better server. It is an aggregator with **no server** — and getting there took a set of mechanisms that did not exist before. Eighteen of them carry the design. Each is listed here with what it is, the mathematics that defines it, why it matters to the person trading, and the section that specifies it in full.

I-1 · Quote ≡ Execution

What. The number you are quoted and the number that binds your trade come from one evaluator — the Core's dispatcher Ω — run once to plan and again inside the executing frame to settle. There is no second pricing model anywhere in the system.

Math. For every leg ℓ the planner and the executor evaluate the same function $\Omega(p_\ell, x_\ell)$ on the same public state; the route floor is re-derived from that in-frame value (E15, E16), never read from the route.

Why it matters. Every aggregator exploit class that lives in the gap between "what the server said" and "what the chain did" has nowhere to live. Measured: drift-free quotes settle at exactly their prediction **64 times in 64**, on mocks and on live Base. (§3, §11, §12)

I-2 · The on-chain solver, free to call

What. `Solver.findBestRoutePlan(tokenIn, tokenOut, amountIn)` is a `view` function. Anyone gets a trustless best route with a free `eth_call` — no API key, no relayer, no signature — and `swapBestExactIn` solves and settles atomically in one transaction.

Math. The whole Solver is the constrained maximisation of the Meta-Equation (E1) over the route space $\mathcal{R}_{\mathcal{D}}$, executed as a deterministic pipeline: derive → rank → probe → filter → allocate → gate → floor.

Why it matters. The routing intelligence of the industry runs on closed servers with no auditable line count. Ours is 29,183 bytes of public bytecode that anyone can re-run and get the same answer. (§2, §10, §11.1)

I-3 · The Meta-Equation

What. The whole aggregator, written as one expression.

Math.

$$R^{\star} = \arg\max_{R} \ \mathrm{net}(R) \cdot \prod_{\ell \in R} \hat{\Psi}(p_{\ell}) \cdot \mathbf{1}[\mathrm{net}(R) \geq \Phi(R)] \cdot \Xi(R)$$

Why it matters. The product form is a safety property: a single failing factor — a leg quoting zero, a pool with no standing, a route under its floor, a hostile hook — sends the route's score to exactly zero. No soft penalties, no tunable weights, no near-miss route sneaking through. (§3)

I-4 · The Eightfold Dispatcher Ω

What. Six venue kinds — Uniswap V2, V3, V4, Algebra, Solidly stable and Solidly volatile — collapse into four mathematical families behind one attribute table and one dispatcher.

Math. Constant product $y = \lfloor x(\mathrm{BPS} - f)\, r_{\mathrm{out}} / (r_{\mathrm{in}}\mathrm{BPS} + x(\mathrm{BPS} - f)) \rfloor$ (E2); concentrated liquidity in Q64.96, $\sqrt{P}' = L\sqrt{P}/(L + x_f\sqrt{P}/Q_{96})$, $y = L(\sqrt{P} - \sqrt{P}')/Q_{96}$ (E3); every dangerous product factored through 512-bit `mulDiv`.

Why it matters. Fragmentation turns out to be a problem of language, not of mathematics. One audited formula per family prices every venue on every chain; a new fork of a known family is a table row, not new code. (§4)

I-5 · Deterministic Derivation $\mathcal{D}$

What. Pools are **computed**, not asked for. The Hub derives each candidate address from the factory, the pair and the init-code hash, and admits only addresses that carry runtime bytecode.

Math. $\mathrm{pool} = \mathrm{addr}_{20}(\mathrm{keccak256}(\texttt{0xff} \,\|\, \mathrm{origin} \,\|\, \mathrm{salt} \,\|\, \mathrm{initCodeHash}))$ (E8).

Why it matters. A factory's answer can be spoofed; arithmetic cannot. Discovery costs no external call per candidate and works identically on every EVM chain, with pool managers and factories as runtime configuration, never hard-coded. (§7)

I-6 · Uniswap V4, entirely in-frame

What. V4 pools have no address and answer no calls. BlazePhoenix derives their ids, **proves** each one live against the PoolManager's storage, reads price and liquidity through `extsload`, walks the tick book across up to sixteen stretches, settles through the singleton — native ETH included — and screens hooks by arithmetic alone.

Math. Storage base slot $=$ keccak256$(\text{abi.encode}(\text{poolId}, 6))$ with $\sqrt{P}$ in bits $[0{,}160)$ and the LP fee in $[208{,}232)$ (E4); the hook gate $(\text{uint160}(\text{hook}) \,\&\, \texttt{0x3FFF}) \,\&\, ((1 \ll 3) \mid (1 \ll 2)) \neq 0 \Rightarrow$ refuse, without a call (E18); the dynamic-fee gate E19.

Why it matters. Derive, prove, price, screen and settle V4 without trusting a single answer from the venue — the deepest V4 integration an aggregator can have, and all of it verifiable on chain. (§5)

I-7 · The Solidly stable solver

What. An exact, fail-closed solver for the quartic stable curve that Aerodrome- and Velodrome-class venues use.

Math. Invariant $k(x, y) = x^3 y + x y^3$ (E5), solved by a bounded Newton iteration with a one-unit bump against the taker and a 64-step cap (E6), after decimal normalisation to 18 places (E7).

Why it matters. Stablecoin and liquid-staking flow clusters on these curves, where they deliver almost the full input as output. The solver quotes them exactly — and where a bound cannot be proven it returns zero rather than a guess. (§6)

I-8 · The Monoslot

What. A pool's entire routing state — activity, depth class, fee, kind, flags, recency — packed into **one 256-bit word**.

Math. $s = a \mid f \ll 8 \mid k \ll 32 \mid r \ll 40 \mid (c \wedge \texttt{0xFFF}) \ll 48 \mid b \ll 60 \mid \ldots \mid B_{\text{last}} \ll 224$ (E9).

Why it matters. One `SLOAD` ranks a pool; one `SSTORE` over one `SLOAD` keeps it current. And the word contains no price, so the registry's memory is structurally incapable of mispricing a trade. (§8)

I-9 · The Self-Healing Registry and the Vitality Field Ψ

What. The protocol learns liquidity by trading it. Sixteen slots per pair, a fitness score earned from real swaps, decay by arithmetic, and a **pair-proof**: a pool proves which tokens it trades before it is stored.

Math. $\Psi = V \cdot 2^{b} \cdot (1 + 0.25\,\mathbf{1}_{\text{bridge}}) \cdot (1 + 0.05\,\mathbf{1}_{\text{conc}})$ (E10); activity halves every 24,576 seconds of silence, $V = \text{swapCount} \gg \lfloor (t - t_{\text{last}})/24{,}576 \rfloor$ (E11); depth class $b = \min(15, \lfloor \log_{10}(d/10^{15}) \rfloor)$ (E12); a newcomer displaces the weakest only past a margin, $2^{b(d_{\text{new}})} > \Psi_{\text{worst}} + \lfloor \Psi_{\text{worst}}/4 \rfloor$ (E13).

Why it matters. No keeper, no cron, no curator deciding which pools matter: the registry heals itself from the flow users were already sending. (§9)

I-10 · The Capital-Anchored Split

What. Price quality and pool depth are measured separately. A tiny probe measures each pool's marginal rate; a **believability band** anchored on capital discards outliers; survivors share the order in proportion to measured depth.

Math. Centre = the depth-weighted median — the first r_i at which $\sum_{j\le i} d_j \ge \lceil \sum d/2 \rceil$ — with band $[0.95, 1.05]\times$ centre (E25); the split survives only if $\text{out}_{\text{split}} \ge \text{out}_{\text{single}} \cdot (1 + 25/10^6)$ (E14).

Why it matters. To move the anchor, an attacker must out-deposit half the pair's real liquidity — capital at risk, not a free donation. The standing guarantee: **a split never pays less than the best single pool.** Measured against an exhaustive grid on Base, the Solver's route sits within 0.13 bps of optimal. (§10)

I-11 · The Iron-Law floor Φ

What. A minimum output re-derived **on chain, at execution**, from measured price impact and route concentration — independent of anything the caller supplied.

Math. $\texttt{floorBps} = \max(9{,}600 - \texttt{legShv} - \min(\delta, 10^4) - \sigma_{\ln}/10^{14},\ 8{,}000)$ (E15), with the concentration shave $\texttt{legShv} = \sum_{\text{hops}} \lfloor 200\,((\sum a)^2 - \sum a^2)/\sum a^2 \rfloor$ (E23); delivery must clear $\texttt{effMin} = \max(\texttt{userMinOut},\ \texttt{route.singleOutFloor},\ \lceil \texttt{finalHopQuote} \cdot \texttt{floorBps}/10^4 \rceil)$ (E16).

Why it matters. Caller data can only tighten protection, never loosen it. On a 1 %-of-depth trade the floor caps a sandwich at about 2.7 % of the trade and **refuses** beyond — the attacker's round trip turns negative past the edge. (§11.5, §14.3)

I-12 · Measurement at every seam

What. The Router never believes a transfer happened; it measures. Every pull, every hop's arrival and every delivery is a real balance delta, from the first instruction of the swap to the recipient's wallet.

Math. Each hop is rescaled against its measured arrival; fee-on-transfer tokens on constant-product legs are priced on the difference the pool actually sees, the exact quantity the pool's own invariant will check.

Why it matters. Tokens that tax, rebase or misreport cannot push the Router into promising what it does not hold — and the Router holds **nothing at rest** between transactions (INV-1). (§11.2, §11.4)

I-13 · The One-Callback Doctrine

What. One `fallback` answers the flash-accounting callback of every V3-shaped venue — Uniswap V3, Algebra, Pancake V3, Sushi V3, the concentrated Solidly forks — with the in-flight leg context held in EIP-1153 transient storage.

Math. The owed token is read from the signs of the two deltas; payment is bounded three ways — only the committed pool, only while a leg is in flight, never more than the leg's recorded budget (INV-10).

Why it matters. One small, triple-guarded entry point replaces a zoo of per-fork callbacks — less bytecode, less surface, one thing to verify. (§11.4)

I-14 · The Surplus Rule and immunity by exhaustion

What. A 0.28 % fee, compiled in, charged once on the first bridge coin the Router holds — and **no fee term that grows with the gap between quote and delivery.**

Math. $\text{fee} = \lceil 28 \cdot \text{base}/10^4 \rceil$, split $t_1 = \lfloor 3{,}000 \cdot \text{fee}/10^4 \rfloor$, $t_2 = \text{fee} - t_1$ (E17); the regime predicate $\text{feeHop} = \min\{h : \text{isBridge}(\text{hops}[h].\text{tokenIn})\}$ (E26). A route that avoids every bridge pays on every hop, so no dust prefix can carry the fee away.

Why it matters. Everything the pools deliver above the quote reaches you in full. The fee cannot be raised against a trader or lowered for a favoured integrator — nobody holds that key. (§13)

I-15 · The byte–gas exchange rate ρ*

What. The four doors of the Router are an affine cost model made into an interface: solve on chain, or carry a pre-solved route in calldata — whichever is cheaper on that chain.

Math. Carry the route in calldata iff $p_{\text{byte}} < \rho^*$, with ρ^* measured at 66.9–105.5 gas per byte by route shape (E22).

Why it matters. Every chain has a price for bytes and a price for computation. BlazePhoenix measures the exchange rate between them and hands the trader the cheaper door. (§17.6)

I-16 · The anti-scam projector Ξ

What. Any pool whose V4 hook declares delta-altering permissions is removed from the candidate set **before ranking**, without the hook ever being called.

Math. $\Xi(R) \in \{0, 1\}$ in the Meta-Equation, evaluated by the E18 bit test on the hook's address.

Why it matters. A hostile hook never gets the chance to lie: it is never asked. Three independent layers — address bits, a codehash-pinned allow-list, and Ξ — stand between a trader and a malicious venue. (§5.5, §14.2)

I-17 · The One-Way Door

What. Administrative authority ends irreversibly, in bytecode. `renounceControl` writes a flag that no code path can unset; after it, only a grow-only curator tier remains.

Why it matters. "We won't touch it" becomes "we can't touch it." The fee, the split and the floors were never settable to begin with (INV-4); after the door closes, nothing else is either. (§14.7)

I-18 · Metrological Design and the published denominator

What. The discipline behind all of the above — measure every coordinate that can be measured, make the rest monotone so a false input hurts only its supplier — and a verification apparatus that reports **every number beside its denominator**.

Math. 1,609 declared tests across 236 files; 268 of 268 curated mutants killed, each paired with the one test that must fail; covering arrays of strength 2 (258 of 258 pairs) and 3 (1,636 of 1,636 triples) over the regime factors; 14 + 5 metamorphic relations; three optimiser builds that agree bit for bit; symbolic proofs in Halmos and the Certora Prover.

Why it matters. Most projects publish a test count. BlazePhoenix publishes how its evidence could be gamed and what it measures against — and recomputes it on every commit. (§2.3, §15)

The scoreboard

MEASURED PROPERTY	RESULT
Drift-free quotes that settle at exactly their prediction (240-sample study)	**64 / 64**
Venue pathologies × doors that settle or refuse with a selector of ours — never a third way	**20 / 20**
Curated mutants killed	**268 / 268**
Control actions whose refusal is exercised by exact code	**31 / 31**
Quantities computed in two places that a named test ties together	**7 / 7**
Pairs of regime-factor values exercised by construction	**258 / 258**
Triples of regime-factor values exercised by construction	**1,636 / 1,636**
Optimiser builds of the quote maths in agreement (runs 1 / 300 / 20,000)	**3 / 3**
Preview → delivery across four chains	**0–1 bps**
Solver route vs. an exhaustive grid on Base	**within 0.13 bps**
Largest sandwich the floor admits on a 1 %-of-depth trade, then refuses	**≈ 2.7 %**
Credited external security researchers · Critical findings	**25 · 0**

§1 The problem: a quote is a promise someone else computed

Imagine selling a car in a city with twenty dealers. A good broker visits all twenty, finds the best price, and perhaps splits the sale across two of them. Most on-chain "brokers" check prices on their own database — possibly stale — and show you a result. You cannot verify that they looked at all twenty, or that they showed you the best one. You can only verify that they have, so far, been honest.

Every aggregator whose documentation we read — 1inch, CoW Swap, Velora (ParaSwap), KyberSwap, OpenOcean, Jupiter and UniswapX, as documented on 2026-09-07 — works this way: a service computes the route off-chain and the contract executes the route it is handed. The contract does not choose; it checks. That is sound engineering — it is why those systems are fast and cheap — and it has one consequence rarely stated plainly: **the strongest claim such a system can make is that its operator is honest.**

The gap is not usually fraud. It is drift, from three ordinary sources. The quote was computed against reserves that moved before the transaction landed. The search was only as exhaustive as the operator's budget. And every correction between quote and fill — a slippage buffer, a fee on the way — was applied by code the user did not read, to a number the user cannot reproduce. None of that requires bad intent. All of it is invisible.

§1.1 The fragmentation is real, and shallower than it looks

A trader who wants the best price faces a paradox. The liquidity exists, spread across dozens of decentralised 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 best execution is the cost of speaking every exchange's language at once, and of trusting every translator between the price shown and the trade that lands.

The incumbent answer integrates venues one at a time: a connector for Uniswap, another for the Solidly forks, another for each concentrated-liquidity variant, each with bespoke quoting, bespoke address lookups, bespoke execution. The integration surface grows linearly with the number of venues, and every connector is a fresh place for a pricing defect or a silent mismatch between the number quoted and the number executed.

The fragmentation is shallower than it appears. Beneath the surface diversity, the automated market makers that hold most 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 over a piecewise-constant active liquidity. The Solidly family obeys a quartic, $x^3y + xy^3 = k$, on its stable pairs and constant product on its volatile ones. There are a handful of shapes, not sixty.

If the shapes are few, 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 shape once, correctly, and dispatches every venue to the right one collapses the integration surface from linear to constant. Adding a venue stops being engineering and becomes configuration: assign a kind and an address-derivation mode, and the existing mathematics prices it, the existing router settles it, the existing registry learns it.

The central claim of the design: every venue BlazePhoenix routes through is a parametrisation of one of a small family of output functions — four working families across a six-kind taxonomy (§4) — held once, in a single stateless library, and never redefined. Adding a venue means assigning a kind and a derivation mode, not writing new pricing code.

§1.2 The off-chain escape hatch

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. A web service produces the price, a database holds the liquidity graph, a privileged operator can change the rules, and the chain becomes a settlement venue for decisions made elsewhere, by software the user cannot see and cannot verify.

That arrangement reintroduces exactly the trust the technology was meant to remove. An off-chain quote can differ from the on-chain fill, and the user cannot prove that it should not have. A keeper can go dark and the router goes blind. A proxy admin can, in one transaction, turn an honest contract into a hostile one.

CONCERN	OFF-CHAIN-SOLVED AGGREGATOR	BLAZEPHOENIX-DEX
Quoting	Off-chain servers or a solver auction	One on-chain dispatcher, shared by the preview and by in-frame execution
Liquidity discovery	Keeper bots index and push lists	Self-Healing Registry — liquidity learned by trading it — plus deterministic discovery
Route selection	Proprietary off-chain optimiser	On-chain Solver, reproducible by anyone from public state
Upgrade surface	Admin-controlled proxy	Stateless Core; no proxy, no upgrade path, no `selfdestruct`
Privileged power over funds	Often present	The Router holds nothing at rest; no function moves user principal
Cross-chain consistency	Per-chain backend logic	One bytecode per chain
Trust assumption	The operator's honesty	The user's own minimum, which the contract refuses to accept as zero

BlazePhoenix rejects both halves of the bargain. It refuses the linear integration surface by showing the surface is not linear, and it refuses the off-chain escape hatch by keeping the entire decision — discovery, quoting, scoring, routing and settlement — inside contracts anyone can read that produce the same answer on every chain. The cost is real: quoting a concentrated pool on-chain is more expensive than reading it from a database. What the cost buys is the removal of one class of failure — the silent, unprovable divergence between what a server promised and what the chain delivered. Every design decision that follows is an instance of that trade.

§2 The answer and the discipline

The thing that chooses is the thing that trades.

BlazePhoenix-Dex has an entry point that takes six values — the two tokens, the amount, your minimum, a recipient, a deadline — and works out the route inside the transaction that executes it, from pool reserves that are public state: which venues, how much to each, in what order. Because the route was derived from public state by public code, anyone can re-derive it. Because the number that binds at settlement is re-derived by the same dispatcher, in the same frame, from the same live state, the quote and the execution are not two artefacts to reconcile; they are one computation, and what separates a preview from a fill is time, not a second model.

Three further entry points accept a pre-computed route from calldata, for callers who want the cheaper path and are willing to supply their own answer; the same measurement, floor and settlement machinery applies to all four doors (§11). We do not claim the off-chain approach is wrong, and we do not claim to be cheaper. We claim a different class of guarantee — auditable rather than promised — for the user who wants it and will pay gas for it.

Four decisions in this design are, as far as we can establish, unusual in the field, and each is stated with the boundary that keeps it honest.

- **Addresses are derived, not fetched.** Wherever a venue family's deployment is reproducible, a pool's address is computed by CREATE2 arithmetic — a theorem anyone can recheck from three public inputs — rather than read from a factory's storage; where the arithmetic is not sound, the protocol asks, and an asked pool proves its own pair before it is listed (§7).
- **A pool's routing state is one word.** The Monoslot packs everything needed to rank a venue into a single 256-bit slot and deliberately contains no price, so the registry's memory can bias which pools are asked but is structurally incapable of pricing a trade (§8–§9).
- **The floor is re-derived from the trade that actually happened.** The Router measures each leg's realised impact and each hop's realised concentration inside the executing frame and recomputes its floor from those measurements, so caller-supplied fields can tighten protection and never loosen it (§11).
- **Discovery is deployer-blind where venues have no addresses.** For the Uniswap V4 singleton, pool identifiers are derived from public parameters and then proven live against the manager's own storage — a proof nobody else can write (§5).

None of these removes the residual every aggregator carries — state can move between signature and inclusion — which is why the minimum you set yourself is mandatory, and final: every door refuses a zero minimum with `RouterE(10)` before a token moves.

§2.1 Invariant-driven design

Behind the answer sits a method. We call it invariant-driven design: the protocol is specified not as a sequence of operations but as a set of properties that must hold in every reachable state, with the code organised so that those properties are structural — enforced by the shape of the code, not by the vigilance of the programmer.

Most software is written imperatively, and correctness is an emergent hope. For ordinary software that is acceptable, because failures are recoverable. 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 is hunted; and a single arithmetic slip is an irreversible loss, not a crash.

An invariant is a property that is true regardless of how the system reached its state. *The Router holds no funds between transactions. A registry pair holds at most sixteen pools, and eviction keeps the fittest sixteen. A settlement that paid no protocol fee does not exist.* Three principles put such statements at the centre:

PRINCIPLE	STATEMENT	WHERE IT APPEARS
Compute, don't trust	Where a value can be derived or proven, never fetch or assume it.	$\mathcal{D}$ computes pool addresses; the Router re-derives fee base and floor in-frame from measured balances.
One source of truth	Every mathematical primitive is defined once and imported; never restated.	The Core holds all pricing, derivation, floor and scoring mathematics; Hub, Solver, Router and Quoter import it and none redefines a primitive. A shared-quantity register (§15) grades every quantity with two producers.

PRIN-CIPLE	STATEMENT	WHERE IT APPEARS
Fail closed	When a property cannot be guaranteed, halt rather than proceed on a guess.	The Solver returns no route rather than a fabricated one (`SolverE(5)`); the stable solver returns zero rather than a saturated quote (§6); the registry's coherence guard reverts a structurally impossible venue before any state is written (§9).

A fourth principle governs custody: **no privileged path to funds.** The Solver and Quoter are read-path contracts; the Router is the only contract that moves money, it moves it atomically under the user's own bound, and every administrative power that could redirect it can be permanently surrendered through the One-Way Door (§14).

§2.2 Why this discipline fits this medium

Each property of the environment that makes imperative code dangerous is a property invariant-driven code turns to advantage. Immutability rewards provability: deployed code cannot be patched, so the value of an up-front proof is correspondingly high. Composability rewards a single source of truth: a pricing function may be invoked inside a vault its authors never anticipated, and with one Core every caller gets the same answer. Adversariality rewards failing closed: an attacker needs only one path the author did not consider, and a system that halts when its assumptions break denies them the improvisation. Value at stake rewards legibility: a protocol expressed as five named operators over one mathematical core can be checked line by line by people who did not write it.

The discipline is not free, and the trade is deliberate: proving an overflow bound takes longer than assuming it away; routing every product through a full-precision multiply-divide costs gas a naïve multiply would not; deriving an address costs engineering that fetching one skips. That is more cost before deployment in exchange for removing the failures that cannot be recovered after it.

§2.3 Metrological Design: one morphism, measured coordinates, monotone inputs

Invariant-driven design says what must always hold. Three further rules say how the code that upholds it is *shaped*, and together they form a method we have not found named elsewhere in the field, so this paper names it: **Metrological Design** — building a contract as an *instrument*. Its thesis in one line: *code that believes is attack surface; code that measures is defence.*

One morphism, evaluated at a point. The aggregator does not contain an integration per venue; it is the evaluation of one parametrised pricing morphism at a point of the space (venue shape, chain, fee model). Expressed as branches, half a dozen live venue kinds with their own fee conventions multiply into hundreds of pairwise combinations — multiplicative bytecode, multiplicative audit surface, multiplicative drift. Expressed as a morphism, the same coverage is a handful of *state shapes*, one stateless Core, fee conventions carried in their native units, and registration tables — and the bytecode grows *additively*: a new venue is a row, not a branch. The rule that guards the collapse is `KIND → SHAPE`: a venue family is *data* — a kind assigned to an existing shape and a derivation mode — never a new code path that adversarial input gets to select. The Core states the shapes as a bit table (`THETA_ATTR` , §4), so "does this kind read reserves?" is a shift and an AND, and there is no default branch for anyone to forget. The collapse has a brake: there is no single universal formula.

Two closed forms are evaluated exactly — constant product and concentrated liquidity — and for every curve where replication would be a liability, the venue's own bytecode is asked (§6). The morphism collapses *membership*, never *verification*.

Coordinates are measured, never believed. The contract never acts on a modelled, nominal or caller-supplied value where an on-chain measurement exists — and no permissionless datum ever chooses which code runs. Data supplied by an untrusted party — a route, a declared fee, a claimed depth, a pool id — are *coordinates*: places to point the instrument, which the contract then reads for itself, fail-closed. Every load-bearing defence in this paper is an instance: the post-pull balance measured instead of the nominal amount (§11), the fee base taken from a measured balance delta instead of from calldata (§13), the believability band anchored on a depth-weighted median of measured depths instead of on a raw balance a donation can inflate (§10), the capacity clamp cut against measured holdings (§10), the V4 fee gate reading live state instead of trusting a sentinel (§5), proof-of-existence discovery reading the singleton's own storage (§5), and an asked pool proving its own `token0` / `token1` before discovery lists it (§7). The repository publishes the classification of every caller-writable field under this rule: 23 fields, 14 confirmed against an observation, 5 steering, 4 declared with the reason (§15).

Where measurement ends, monotonicity begins. Some numbers can be neither measured nor re-derived — a caller's floor, a caller's declared scale, the caller's own minimum. For these the design accepts the number and shapes the function so that lying can only hurt the liar: the route's attested floor can only *raise* the enforced floor (§11); a leg's attested quote is lifted to the measured quote when it under-covers it (§11); declared scaling is capped by measured arrival, so overstating cannot route funds that never existed; and the one number the protocol cannot choose — your minimum — is mandatory. A monotone input needs no trust, so it can be computed off-chain, by anyone, for free, and carried in calldata — which is how this protocol moves work off-chain *without moving trust off-chain*, and why the Router can offer a solving door and three calldata doors with the same guarantees.

Metrological Design governs the engineering around the artefact too, because an instrument is only as good as its calibration. Safety margins are priced from measured hazards: the split gate sits at 25 ppm because the leg break-even measured 3 ppm (§10); a planning haircut that outweighed its measured hazard 27× was retired (§10). External comparisons carry their method's error bars or are not printed (§17). And the measuring apparatus is itself measured: each assurance instrument was calibrated against a defect already known by another route before its number was printed (§15). That is the whole theory; the rest of this document is its practice.

§3 The Meta-Equation and the dependency budget

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

$$R^{\star} \;=\; \arg\max_{\,R \,\in\, \mathcal{R}_{\mathcal{D}}(t_{\text{in}},\, t_{\text{out}},\, x)} \;\; \text{net}(R) \cdot \prod_{\ell \in R} \hat{\Psi}(p_{\ell}) \cdot \mathbf{1}\big[\,\text{net}(R) \geq \Phi(R)\,\big] \cdot \Xi(R) \tag{E1}$$

$$\text{net}(R) \;=\; \sum_{\ell \in R} \Omega(p_{\ell},\, x_{\ell}) \;-\; f_{\text{proto}}$$

Here $\mathcal{R}_{\mathcal{D}}$ is the route space over pools located by the Deterministic Derivation $\mathcal{D}$ and the registry; $\Omega(p_\ell, x_\ell)$ is the dispatcher's output for leg ℓ pushing x_ℓ through pool p_ℓ; f_{proto} is the protocol fee; $\hat{\Psi}(p_\ell)$ is the leg's normalised vitality; $\Phi(R)$ is the Iron-Law floor for the route's measured impact and concentration; and $\Xi(R) \in \{0, 1\}$ is the anti-scam projector.

OPERATOR	NAME	ROLE IN THE EQUATION	SECTION
Ω	Eightfold Dispatcher	Prices each leg — one branch per pool kind — and reports a token-denominated depth for the split allocator	§4
$\mathcal{D}$	Deterministic Derivation	Locates the pools the route uses, by computed address where the family permits it, by proven pool id for the V4 singleton	§5, §7
Φ	Iron-Law floor	Projects the route onto the feasible set: a floor under net output, re-derived in the executing frame	§11
Ψ	Vitality Field	Weights each pool by earned quality, read from its packed on-chain state	§9
Ξ	anti-scam projector	Values any route that is unsafe to surface at exactly zero	§5, §14

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*; $\mathcal{D}$ answers *where the pool lives*; Ξ answers *whether the route is safe to show*. The product form means a single failing leg collapses the route's score — a leg that quotes zero, or a pool with no standing, zeroes the product rather than discounting it — and the indicator hard-zeroes any route whose net output falls below the floor. No soft penalties, no tunable weights, no near-miss route sneaking through.



FIG. 1 The Meta-Equation as the Solver's pipeline: candidates from $\mathcal{D}$ and the registry, ranked and truncated by Ψ, quoted by Ω, allocated across legs, checked against Φ and Ξ; the Router re-evaluates Ω and Φ in the executing frame.

On chain, the Solver realises the equation as a pipeline (`findBestRoutePlan`): candidates per pair are gathered by the registry and, when the registry is not fresh, by $\mathcal{D}$; ranked and truncated by Ψ to `MAX_CANDIDATES = 8`; probed and quoted by Ω; filtered by a believability band; allocated across legs; and the assembled route carries the floor Φ attests. The Router then re-evaluates Ω and Φ on the amounts that actually arrived. The claim the equation makes — the one an auditor can hold the code to — is that no step of that pipeline consults anything outside the five operators. If a quantity is not an argument to one of the five, it cannot influence the route: no venue's marketing, no off-chain reputation, no operator preference.

§3.1 The dependency budget

Instead of adjectives — *decentralised*, *serverless*, *trustless* have stopped carrying information — here is the checkable list: everything that must exist, and keep working, for one trade to settle.

DEPENDENCY	TYPICAL OFF-CHAIN-SOLVED AGGREGATOR	BLAZEPHOENIX-DEX
A service that computes the route	Required	Not used — the Solver is a `view` function any node serves
A price oracle or feed	Varies	None — no feed call exists in the five contracts
A solver network or auction	Common	None

DEPENDENCY	TYPICAL OFF-CHAIN-SOLVED AGGREGATOR	BLAZEPHOENIX-DEX
A keeper, cron or relayer	Varies	None — vitality decays by arithmetic when next read; the registry learns from ordinary trading
An upgradeable proxy	Common	None — no proxy, no `selfdestruct`, no initialiser; the only `delegatecall`s are the compiler's own into the public Core library, fixed at link time
A hosted interface	In practice yes	Convenience only

The last row is where most budgets quietly fail. The hosted interface is a convenience, not a dependency: `swapBestExactIn` takes six values a person can assemble by hand, and an integrator, a script, or a competitor's front end can call it with no relationship to us of any kind. If every server this project operates disappeared tonight, the contracts would quote, route and settle tomorrow exactly as they did today. The cost of that property is that the caller pays for the search in gas; the budget above is the list of things that gas buys out.

Immutability cuts the other way too. A defect cannot be patched — the price of a contract you audit once — which is why defects are disclosed and bountied instead (§21).

§3.2 Three edges

The budget has three edges, stated next to the claim they bound. The interface is hosted, like everyone's, and a compromised front end is the single largest exposure in the threat model: the contract honours whatever non-zero minimum it is handed, so the minimum you set is the one bound no interface can take from you. Venue discovery reads state that earlier transactions wrote — public and permissionless, but not conjured at trade time. And administrative powers exist until renounced through the One-Way Door; after renunciation the control tier — pause, roles, treasuries, wiring, removals — is gone for ever, and only the grow-only curator tier remains (§14). Stateless core, serverless operation, grow-only governance: the table is the claim; the words are its shadow.

§4 The Eightfold Dispatcher Ω

Plain: one function that knows how to price every kind of pool. *Precise:* `universalQuote(QuoteCtx, amountIn)` in the Core is the quote dispatcher for the Solver and for the Quoter's preview; it returns the expected output and a token-denominated depth for the split allocator, and a pool it cannot price returns zero and is dropped before anything is sent. The Router evaluates the same Core primitives per leg in its own frame (`_hopScaleImpactAndQuote`), a sibling kept for gas and stack depth, and the Quoter's Exact Pass asks a different question again (§12); the three are recorded together in the repository's shared-quantity register, and their agreement is pinned by named tests rather than assumed (§15). The house name is kept from the eight-kind enumeration the dispatcher shipped with; the enumeration has since grown a ninth member and surrendered three to deliberate excision. The numbering keeps its gaps: a Monoslot recorded yesterday must decode identically for ever.

§4.1 Six kinds, four families, one attribute table

KIND	CONSTANT	FAMILY	PRICING BRANCH	0 ATTRIBUTES	GAS LADDER (SOLVER)
0	`KIND_V2`	Constant product (Uniswap V2, Sushi, forks)	closed form `outV2` over `getReserves`	reserves · pair-verifiable	90,000
1	`KIND_V3`	Concentrated liquidity (Uniswap V3 and V3-shaped forks)	closed form `outV3` over `slot0`	concentrated-at-pool · pair-verifiable	110,000
4	`KIND_V4`	Uniswap V4 singleton	closed form `outV3` over `extsload`	concentrated-in-singleton	180,000
5	`KIND_SOLIDLY`	Solidly (Aerodrome, Velodrome, forks), stable and volatile	ask the pool `getAmountOut`; replicated curve as fallback	reserves · pair-verifiable	90,000
6	`KIND_ALGEBRA`	Algebra (Camelot-class), dynamic fee	closed form `outV3` over `globalState`	concentrated-at-pool · pair-verifiable	110,000
8	`KIND_V4_NATIVE`	V4 pool holding the chain's native currency	same branch as kind 4	concentrated-in-singleton	215,000

Six kinds, four working families: constant product {0}, concentrated {1, 6}, Solidly {5}, and the V4 singleton {4, 8}. The attribute column is not prose: it is the bit table `THETA_ATTR = 0x040A9400A9` in the Core, four bits per kind — *reads reserves*, *concentrated at the pool address*, *concentrated in the singleton*, *exposes* `token0()` / `token1()` — so every membership question in the five contracts ("does this kind read reserves?") is one shift and one AND (`kindHas`, `kindHasAny`), and a kind with no bits fails closed with no default branch for anyone to forget. The gas column is a second table, `THETA_GAS`, read only by the Solver's `estGas`, in units of 5,000 gas per leg.

Kinds 2, 3 and 7 are absent by deliberate excision. The weighted family (3) never produced a routable quote. The two Curve families (2, 7) worked and were removed anyway, in the trust-boundary hardening of August 2026, because supporting them required the Router's only token `approve`, the only depth figure attested by a caller instead of measured from a pool, and the only exception in the Hub's pair-authenticity proof — three attack surfaces, each the last of its class. The retired numbers are never reassigned: `decodeKind` reads the kind from Monoslot bits, so reusing 2 would reinterpret every pool ever recorded under it, and a CI guard fails the build if an excised symbol returns.

The native-currency kind (8) exists because one invariant breaks there and nowhere else. The Router's working assumption — every asset is an ERC-20 with `balanceOf` — fails exactly when a V4 pool's currency is the chain's native asset, and encoding that exception as a *type* makes it visible in the pool record, carried in the route plan, and greppable by an auditor. The pricing needs no branch: the pool identifier derives from the two sorted currencies, and `address(0)` sorts first by construction. The kind earns its place empirically: measured on 2026-08-13 across the chains served, 62.9 % of the ETH-denominated liquidity inside V4 sat in native pools (Arbitrum 99.6 %, Optimism 95.0 %, Base 48.9 %).

§4.2 The closed forms

Constant product. Reserves satisfy $x \cdot y = k$ and a fee f in basis points is taken from the input. The Core computes

$$y \;=\; \mathrm{outV2}(x,\, r_{\mathrm{in}},\, r_{\mathrm{out}},\, f) \;=\; \left\lfloor \frac{x\,(\mathrm{BPS} - f)\,r_{\mathrm{out}}}{r_{\mathrm{in}} \cdot \mathrm{BPS} \;+\; x\,(\mathrm{BPS} - f)} \right\rfloor \tag{E2}$$

with no division until the final step, so the only rounding is the floor of the last quotient, and a fee of `BPS` or more makes the pool unquotable (returns zero). The fee that enters this formula is `effV2Fee(declared)`: a declared fee of zero or above `V2_FEE_CEILING_BPS = 100` is replaced by the canonical 30 bps, so a registration cannot deflate its own quote — and with it the floor that quote anchors — by declaring an implausible fee; the same rule is mirrored on the execution path, so both sides of the trade price with one number.

Worked example 1 — a constant-product quote. A pool holds $r_{\mathrm{in}} = 1{,}000{,}000$ USDC and $r_{\mathrm{out}} = 500$ WETH at the canonical 30 bps; a trader sells $x = 10{,}000$ USDC. In the pool's own units (USDC has 6 decimals, WETH 18) $x = 10^{10}$, $r_{\mathrm{in}} = 10^{12}$, $r_{\mathrm{out}} = 5 \times 10^{20}$:

1. Fee-adjusted input: $x(\mathrm{BPS} - f) = 10^{10} \times 9{,}970 = 9.97 \times 10^{13}$.
2. Numerator: $9.97 \times 10^{13} \times 5 \times 10^{20} = 4.985 \times 10^{34}$.
3. Denominator: $10^{12} \times 10^{4} + 9.97 \times 10^{13} = 1.00997 \times 10^{16}$.
4. Output: $\lfloor 4.985 \times 10^{34} / 1.00997 \times 10^{16} \rfloor = 4.935790 \times 10^{18}$ wei, i.e. **4.935790 WETH** (to six decimals; the contract floors the last quotient to the wei).

At the spot price of 0.0005 WETH per USDC the same 10,000 USDC would fetch 5 WETH; the fill is 1.28 % below spot, of which 0.30 % is the fee. The Core's reserve-based impact measure for this leg is $\mathrm{impactV2Bps} = \lceil x \cdot \mathrm{BPS} / (r_{\mathrm{in}} + x) \rceil = \lceil 10^{14} / 1.01 \times 10^{12} \rceil = 100$ bps — the number the Iron-Law floor consumes in §11.

Concentrated liquidity. V3-shaped pools store no reserves; they store a square-root price $\sqrt{P}$ in Q64.96 fixed point and an active liquidity L. Within a tick, for token0 → token1 with fee f in parts per million and $x_f = \lfloor x\,(10^6 - f)/10^6 \rfloor$:

$$\sqrt{P}' \;=\; \frac{L\sqrt{P}}{L \;+\; x_f\sqrt{P}/Q_{96}}, \qquad y \;=\; \frac{L\left(\sqrt{P} - \sqrt{P}'\right)}{Q_{96}} \tag{E3}$$

and for token1 → token0, $\sqrt{P}' = \sqrt{P} + x_f Q_{96}/L$ and $y = L\,Q_{96}(\sqrt{P}' - \sqrt{P})/(\sqrt{P}\,\sqrt{P}')$, the division split in two stages. The naïve computation overflows: $L \cdot \sqrt{P}$ can reach 2^{288}. The Core never forms it — every product is factored through a full-precision `mulDiv` (512-bit intermediate, 256-bit result), so each stage provably fits, and a step that would move the price the wrong way or to zero returns zero rather than a number. Algebra pools (kind 6) share the branch and differ in one respect the branch handles: their state and their dynamic fee live in `globalState`, read once (§5.4).

The formula is exact within a single tick and is not exact across tick boundaries, where the single-tick form overestimates. The dispatcher does not replicate tick-crossing arithmetic: replicating a venue's internal mathematics is a structural liability, because any divergence between copy and original surfaces as a mispriced quote. Two things stand in its place. The Core carries a boundary clamp (`sqrtBoundary`) that truncates the step at the current range's edge — exact for what fits inside the range and strictly below for the rest, never

above — and the design places it on the *promise* side, not on ranking, because clamping only the families that have ticks would compare venues under two conventions. In the tree this paper measures the clamp is defined but not yet wired: `Router._v4LegQuote` and the Core's ranking arm both call the unclamped single-tick form, so a concentrated leg that leaves its range is bounded only by the per-leg floor of §11.3 and the hop budget,, with the Iron-Law floor as the backstop. And at execution the Iron-Law floor is enforced against a quote the Router re-derives in-frame from the amounts that actually arrived, so a tick-boundary overestimate costs a route its ranking accuracy, never the user the floor. The Quoter's Exact Pass (§12) gives an integrator the pool's own number when it wants one.

Depth. Every branch also returns a depth in token units — the shorter reserve side for reserve families (`shortSide18`), and for concentrated families the virtual reserve $L/\sqrt{P}$ or $L\sqrt{P}$ that L represents at the current price (`depthFromL18`), so a V3 pool and a V2 pool of equal depth compare equal in the allocator instead of differing by a factor of $\sqrt{P}$.

§4.3 Two fee units, carried natively

Constant-product and Solidly venues quote fees in basis points; concentrated venues in parts per million. The dispatcher carries each in its native unit and never cross-converts, which removes a class of rounding failure at the price of two constants instead of one. Two venue families know their fee only at execution time — V4 dynamic-fee pools and Algebra — and both are handled by measuring the fee from the same state read that supplies the price (§5.4).

§4.4 One dispatcher, two surfaces

KIND	QUOTER PREVIEW	IN-FRAME (SOLVER'S PLAN; ROUTER'S RE-DERIVATION)
V2 (0)	closed form `outV2`	closed form `outV2`
V3 / Algebra (1, 6)	closed form `outV3` over `slot0` / `globalState`; Algebra's live fee measured from the same read	same
Solidly (5)	ask the pool; replicated curve × 9,800 / 10,000 as fallback	same
V4 / V4-native (4, 8)	closed form `outV3` via `extsload`, fail-closed fee gate (E19)	same

What separates a preview from a fill is therefore time: state that moved between the preview block and the execution frame. §12 measures exactly that gap on 240 samples.

§5 Uniswap V4 end to end

Uniswap V4 is not a new pool contract; it is a new shape of venue, and it invalidates four assumptions every router built between 2020 and 2024 rests on. Pools have no addresses — they are entries in one singleton's storage, keyed by a hash. Arbitrary code runs inside the swap — a hook executes at defined points of the set-

tlement the router is performing. Fees need not be constant. And pools may hold the chain's native currency, which has no `balanceOf`. This section answers all four structurally, so that a V4 leg is — to the Solver, to the floors and to the trader — indistinguishable from any other leg.

§5.1 Locating a pool that has no address: derive, then prove

The Deterministic Derivation $\mathcal{D}$ computes addresses (§7), but a V4 pool has no address to compute. What it has is an identifier — the keccak hash of its `PoolKey` — so mode 9 of the derivation table derives *poolIds* rather than addresses, and then does something no address derivation can: it proves the pool exists, by reading the PoolManager's own storage for that id and requiring a non-zero square-root price and non-zero liquidity. Nobody can forge that proof, because nobody but the manager writes that storage. A fabricated id resolves to zeros and is discarded.

Worked example 4 — a V4 pool identifier from its key. The canonical id is $\text{poolId} = \text{keccak256}(\text{abi.encode}(c_0, c_1, \text{fee}, \text{tickSpacing}, \text{hooks}))$ (`computeV4PoolId`), five 32-byte words. Take the native ETH / USDC pool on Base at the 0.05 % tier: $c_0 =$ `0x0000...0000` (native currency sorts first), $c_1 =$ `0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913` (USDC), fee $= 500$, tickSpacing $= 10$, hooks $=$ `0x0`.

1. `abi.encode` lays out 160 bytes: the two currencies left-padded to 32 bytes each, then `500`, then `10` (an `int24`, sign-extended to 32 bytes), then the zero hook.
2. poolId $=$ `0x96d4b53a38337a5733179751781178a2613306063c511b78cd02684739288c0a`.
3. The registry's `pool` field for a V4 row is the id's low 160 bits, `0x781178a2613306063c511b78cd02684739288c0a` — an address with no code, which is why V4 rows are never asked `token0()`, never read for a balance, and prove themselves by re-derivation instead (§9).
4. The pool's storage base is $\text{keccak256}(\text{abi.encode}(\text{poolId}, 6)) =$ `0x13a3a697...73b2893d`: slot 6 of the manager is the `pools` mapping. The Core reads word 0 at that slot and word 3 at base $+ 3$ (`v4SqrtAndLiq`, selector `extsload(bytes32)` = `0x1e2eaeaf`).
5. Word 0 packs $\sqrt{P}$ in bits [0, 160), the tick in [160, 184), the protocol fee in [184, 208) and the LP fee in [208, 232); word 3 carries the active liquidity in its low 128 bits. If either $\sqrt{P}$ or L is zero, the pool does not exist for this protocol.

The wrapped sibling of the same pair — $c_0 =$ `0x4200...0006` (WETH), same tier — has id `0x90333bb0...ebd943a0`: a different pool with different depth, which is why the scan probes both keys and probes the native one first (below). The layout in step 5 is not taken from documentation: the same pool read through this layout and through the canonical `StateView` contract returns the same numbers on a mainnet fork, and the identity is what the Core's comment cites.

The cost ladder. The scan that produces candidate ids is a ladder, cheapest rung first, because a naïve sweep over every fee-and-spacing combination would be an unacceptable gas bill on the solving path:

RUNG	WHAT IT TRIES	COST WHEN IT HITS
1	The tier pattern already learned for this token pair	one probe — the steady-state path
2	A provenance check before paying for a cold scan	one read

RUNG	WHAT IT TRIES	COST WHEN IT HITS
3	The canonical Uniswap tiers, batched through `extsload(bytes32[])`	one call for the batch
4	The `(fee, spacing)` pairs the registration row declares	bounded by the registration
5	The cold-start generator grid — fee $= 10{,}000\,j$, spacing $= 100\,j$, j descending from `V4_GRID_MAX = 99`	at most `V4_GRID_PROBES = 40` probed

The whole scan stops at `V4_CAP = 8` proven pools. Two properties fall out of the construction rather than from a check: a hooked pool can never be emitted by this scan — it derives *hookless* ids, and a hooked pool's hookless id does not exist in the manager, so it fails the liveness proof before any policy question is asked — and duplicate ids from different tier guesses collapse to one candidate, so one pool cannot saturate the funnel by aliasing. Probe order is itself a policy, because the cap is spent by whatever is probed first: a pair against the chain's native asset has two keys in V4, native and wrapped, and the native key was measured to be the deeper one on the chains served when the ordering was decided (3.76× on Base, 292× on Robinhood Chain), so it is probed first.

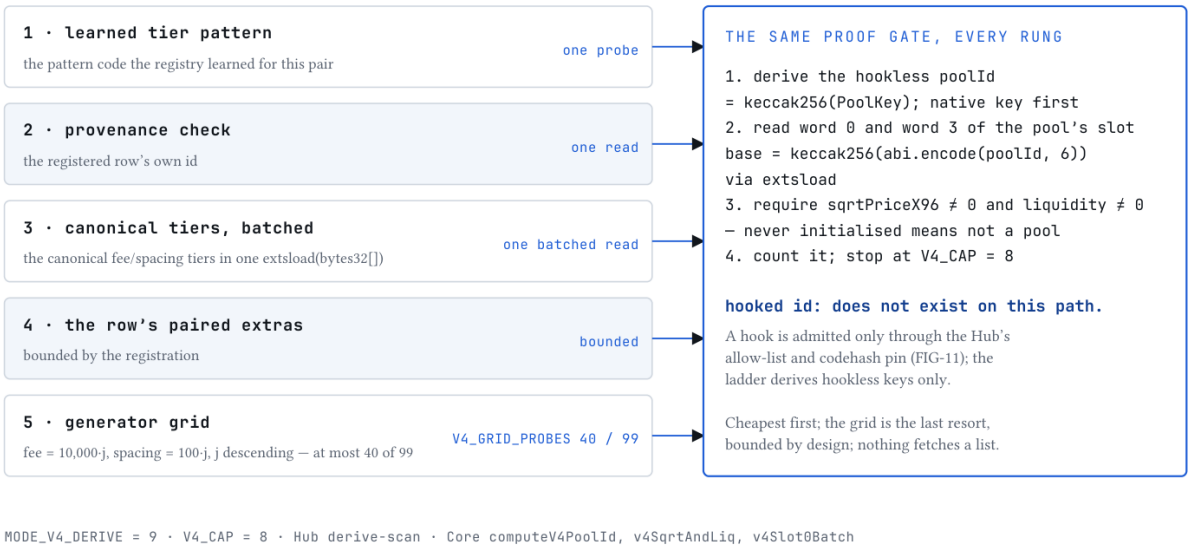


FIG. 2 The V4 derive-and-prove ladder: five rungs from the learned tier pattern to the bounded generator grid, each candidate id proven live against the manager's storage before it counts toward the cap of eight; hooked ids never exist on this path.

The permissionless door. Anyone may call `claimV4(c0, c1, fee, tickSpacing)` and have a hookless pool admitted, because the claim carries its own proof: the id is recomputed from the submitted key and read back from the manager's storage, so a false claim resolves to zeros and admits nothing (`HubE(9)`). Three further conditions gate the door: one side of the pair must be a routable bridge coin, so that a seat in the capped registry is never spent on a pair the router can never cross; the effective fee must be quotable under the fee gate below; and the claim must clear the same admission margin a swap-driven registration clears (§9), with the pool's measured liquidity as its depth. The registry can be taught by the public and cannot be lied to by it.

§5.2 Reading a pool that answers no calls

With an id proven live, the dispatcher needs price and liquidity — and there is no `slot0()` to call. The Core reads the singleton's storage directly by `extsload` as in the example above, and the V4 branch reduces to the concentrated-liquidity closed form of §4.2: the same mathematics, the same overflow discipline. The singleton changes where the state lives, not what the state means. A codeless or non-conforming manager yields all-zero words — fail-closed, never a revert.

§5.3 Settling with a singleton

Execution inverts the usual control flow. Instead of calling a pool and being called back for payment, the Router calls `unlock` on the manager and performs the entire swap inside the callback the manager grants it: swap, then settle every owed currency through `sync → settle → take`, then release the lock. The in-flight currencies are carried across the unlock boundary in transient storage (`TSLOT_V4IN`, `TSLOT_V4OUT`), so the frame dirties no persistent state and the context is provably empty between swaps. The `BalanceDelta` the manager returns is packed — `amount0` in the high 128 bits, `amount1` in the low — and the Router decodes it as such before any settlement arithmetic runs.

Native currency keeps the invariant that matters. The Router is WETH-canonical everywhere — native value entering `swapExactInNative` is wrapped exactly once at the door, and native output is deliberately not implemented: the recipient receives WETH, which no path can silently trap. A native V4 leg unwraps *just in time* inside its own unlock frame — `WETH.withdraw`, then `settle` with value, with ETH received from a native `take` re-wrapped in the same frame — so native ETH never exists at rest in the contract. The one new surface this required is a `receive()` gated as narrowly as the requirement allows: a transient slot (`TSLOT_ETHOK`) holds the single address permitted to send raw ETH to the Router at that instant — the canonical WETH contract during the unwrap, the PoolManager during a native take, zero at every other moment — so bare ETH from anyone else reverts. The native door is fail-closed until `weth` is wired (`RouterE(3)`).

§5.4 The fee that exists only at execution time

A static-fee V4 pool carries its LP fee in its key, where it is immutable. The pool manager may additionally take Uniswap's own manager fee — capped at 1,000 pips, one tenth of one percent, per direction — that the Core's static arm does not add, so a quote on a static key can overstate the delivered output by up to that manager fee once Uniswap's fee controller enables it (it is zero on every chain served at the time of writing); the shortfall lands inside the floor, and the fee base, being measured, ignores it. A dynamic-fee pool carries the sentinel `0x800000` instead, and its real fee lives in `slot0`. Quoting the sentinel at face value would price the trade as free while execution paid the live rate. The gate is three lines in the Core (`effV4Fee`):

$$f_{\text{eff}} \;=\; \begin{cases} f_{\text{key}} & f_{\text{key}} \neq \texttt{0x800000} \\ \texttt{0xFFFFFF}\ (\text{unquotable}) & f_{\text{key}} = \texttt{0x800000} \;\wedge\; f_{\text{proto}} \neq 0 \\ f_{\text{LP}}\ \text{read from}\ \texttt{slot0} & \text{otherwise} \end{cases}$$

(E19)

A dynamic-fee key riding with a non-zero protocol fee returns a fee at or above 10^6, and `outV3` returns zero for it: fail-closed, because the composition of protocol and LP fees is not anchored to our satisfaction, and the measured protocol fee on Base at the time of writing is zero, so the branch costs nothing in practice. The same rule closes the same seam one family over: Algebra pools are dynamic-fee by construction and register

the zero sentinel, so `quoteV3Fee` takes a non-zero configured fee as the truth, a dynamic pool's fee from the `globalState` read that already supplied its price, and — for concentrated-liquidity forks whose fee is keyed by tick spacing and lives on the pool — the pool's own `fee()`, failing closed at `0xFFFFFF` when nothing answers. Static V3 pools take no extra read and quote byte-identically.

One seam remains and is bounded rather than removed: a dynamic-fee hook can override its fee per swap inside `beforeSwap`, so `slot0`'s stored fee is not in every case the executed one. That residual is contained by the Hub's hook allow-list at admission and by the Iron-Law floor re-derived from realised output at execution, so a hostile override lands as a revert or a within-floor shortfall, never a fill below the floor (§14).

§5.5 Hooks: screened by arithmetic, never by interrogation

A hook is arbitrary code running inside the settlement the Router is performing. The intuitive defence — asking the hook what it intends — is not a defence, because a hostile hook lies or re-enters. BlazePhoenix never asks a hook anything. Three independent layers stand instead, cheapest first, and §14 develops them: the hook's permission bits live in the low fourteen bits of its own address, fixed at deployment, and the two bits that permit altering swap deltas are refused by one AND before a token moves (`hookAltersDeltas`, `RouterE(9)`); a hook is routable only while allow-listed and while its runtime code still matches the code-hash pinned at admission (`isHookLive`); and the projector Ξ removes a delta-altering hook from the Solver's candidate set before ranking, so the route is never even representable. The residual is stated with the mechanism: an allow-listed, delta-free hook can still revert mid-swap and waste the gas of trying — the price of refusing, on principle, to call untrusted code to find out what it would have done.

§5.6 What the integration is worth

Every mechanism above is either a Core primitive already used elsewhere, applied to a new state layout, or a genuinely new primitive that generalises beyond V4 — proof-of-existence discovery. Nothing is a per-pool integration, and nothing is a curated list a human must keep current. V4 pools are discovered without a directory, proven against storage nobody else can write, priced by the same closed form as every other concentrated venue, fee-gated by measurement with a fail-closed refusal where measurement is not exact, screened for hostile hooks by immutable address arithmetic, settled through the singleton's own lock protocol, and reachable in native ETH without the Router holding a wei of it at rest. §17 carries the measurements: in the factory census on Arbitrum the V4 family wins more hop-pairs than any other registered venue, and on Optimism the census recorded what is, to our knowledge, the chain's first V4-routed aggregation. We know of no other public aggregator that derives, proves, prices, screens and settles V4 pools entirely inside the executing transaction, native ETH included, with no curated pool list; if you find one, the claim is wrong and we would like to know.

§6 The Solidly solver

Some invariants have no closed form simple enough to reproduce safely, and reproducing them is exactly the replication risk the protocol refuses elsewhere. For such venues the dispatcher asks the pool, through the narrowest honest channel the venue exposes — and holds a replica in reserve for forks that do not answer.

The Solidly family — Aerodrome, Velodrome and their forks — serves correlated assets on stable pairs with a quartic invariant that hugs the diagonal far more tightly than constant product while still admitting divergence:

$$k(x, y) \;=\; x^3 y \;+\; x\, y^3 \;=\; x\, y \left(x^2 + y^2\right)$$

(E5)

Near the balance point this curve delivers almost the full input as output — minimal slippage exactly where stablecoin and liquid-staking flow clusters. Volatile Solidly pairs are constant product, and the Core's volatile arm *is* `outV2` — one producer, which metamorphic relation MR7 pins (§15).

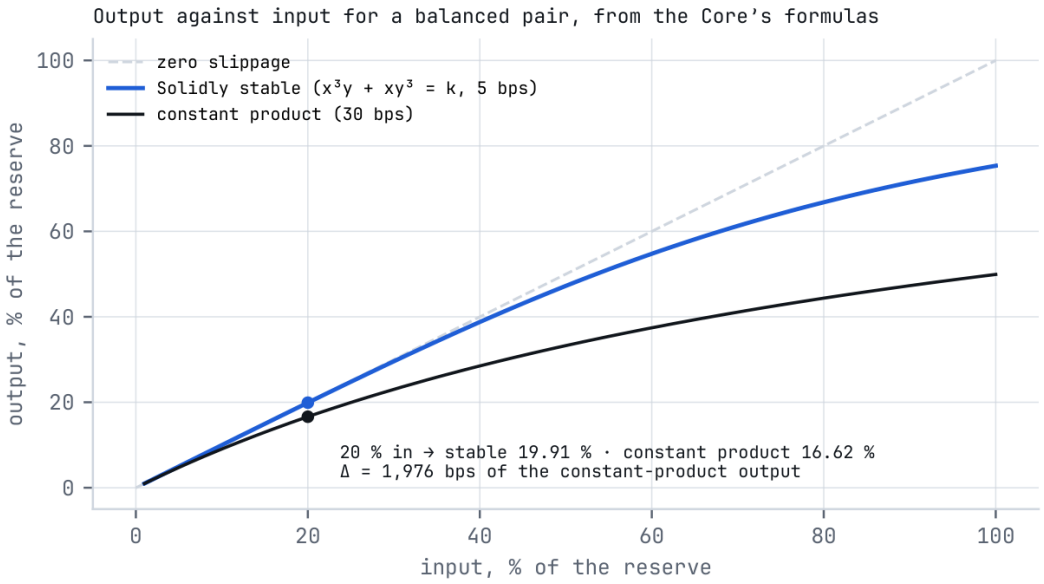


FIG. 3 Output against input for a balanced stable pair: the Solidly quartic against constant product and the zero-slippage diagonal, from the Core's own formulas.

Primary path: ask the pool. `getAmountOut(amountIn, tokenIn)` is computed by the pair's own bytecode — live fee, curve and rounding included — so a swap requesting exactly that figure satisfies the pool's invariant check by construction, and no margin is needed. The trigger for the fallback is an answer of at most 1 wei, aligned across the three channels that ask (Solver, Router, Quoter) after a pool answering exactly 1 once sent two of them down different branches.

Fallback: replicate, then under-ask. When a fork does not expose the selector, the dispatcher reads the live fee from the pair's factory (`getFee(pool, stable)`; a declared fallback fee is believed only within the same 100 bps ceiling as the V2 arm, else 30 bps), solves the invariant on decimal-normalised reserves, and multiplies the result by 9,800 / 10,000 — a 200 bps haircut so that the pool's rounding, which the replica cannot observe, always has slack. The haircut applies only on the fallback and never when the pool answered for itself.

The solve. Given the post-trade input balance x and the target invariant K from the pre-trade reserves, the Core seeks the root of $f(y) = x^3 y + x y^3 - K$, with $f'(y) = x(x^2 + 3y^2) > 0$, by Newton iteration seeded at $y_0 = Y$, the opposite pre-trade reserve (`_solY`):

$$k(x,y_n) < K:\;\; y_{n+1} = y_n + \max\!\left(1,\;\left\lfloor \tfrac{K-k(x,y_n)}{f'(y_n)} \right\rfloor\right); \qquad k(x,y_n) \geq K:\;\; y_{n+1} = y_n - \delta_n,\;\; \delta_n = \begin{cases} \lfloor y_n/2 \rfloor & d_n \geq y_n \\ \max(1,d_n) & \text{otherwise} \end{cases}$$

(E6)

The natural-looking seed — the constant-sum approximation — is wrong in a way that silently corrupts prices: for a pool absorbing an input several times its reserve it lands far above the root on a region where f is extremely steep, the step floors to zero, and the solver pins at a confident, plausible, incorrect number. Seeding at the opposite reserve and stepping in both directions converges for balanced and heavily skewed pools alike. A downward step that would cross zero is replaced by half the iterate; every step forces at least one unit of progress; the iteration exits when successive iterates differ by at most one unit and lands on the invariant-safe side by a final one-unit adjustment against the taker. The loop is capped at 64 iterations and the cap fails closed: the solver returns the seed itself, which the caller's $y \geq Y$ guard maps to a zero quote — the pool is treated as unpriceable, matching how Aerodrome itself refuses rather than misprices. A vanishing derivative takes the same exit.

Decimal normalisation. The invariant is homogeneous of degree four, so when both tokens share decimals the scale cancels and raw reserves work directly. A pair such as DOLA (18 decimals) against USDC (6) carries the mismatch into the quartic, so the Core scales both sides to a common 10^{18} basis before solving and de-scales afterwards:

$$X = r_{\text{in}} \cdot 10^{\,18-d_{\text{in}}}, \quad Y = r_{\text{out}} \cdot 10^{\,18-d_{\text{out}}}, \quad A = \left\lfloor \frac{x \cdot 10^{\,18-d_{\text{in}}}\,(\text{BPS}-f)}{\text{BPS}} \right\rfloor, \quad \text{out} = \left\lfloor \frac{Y - y^{\star}(X+A,\,K)}{10^{\,18-d_{\text{out}}}} \right\rfloor$$

(E7)

Tokens reporting more than eighteen decimals quote zero rather than underflow the scaling. The other token's decimals are read from the pool, not taken from the caller — a single-producer primitive that accepted its caller's coordinate would reopen the divergence it exists to close.

Fail-closed on the whole domain. Four guards decide whether the curve is evaluated at all, and each returns zero rather than reverting, because a revert inside a quote is a library `DELEGATECALL` unwinding the Solver's whole plan for the pair: X and Y must not exceed 3.4×10^{38}; $X + A$ must not either; and `_solKFits` requires both the invariant $k = xy(x^2+y^2)/\text{WAD}^3$ and the derivative $x(x^2+3y^2)/\text{WAD}^2$ to fit a 256-bit word at the seed, which is the largest point Newton visits. That last test is asked without a division — it computes the high word of the 512-bit product and compares it with one WAD (`_mulFitsWad`) — because the earlier form, $a \leq \max \cdot \text{WAD}/b$, itself overflowed on a pool holding under one unit of a token. The N-version lane of §15 found that escape on 2026-09-05; the repair is 35 bytes smaller than the code it replaced and went in red-first, with a 5,000-run fuzz over the full domain, a 60,000-sample sweep of the same arithmetic in Python, and three mutants in the guard.

The measured residual, and the 200 bps margin. The replica was measured against live Aerodrome pools with the pool's own `getAmountOut` as ground truth:

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 %

The balanced pair matches to the integer; the skewed pairs sit within 0.5 %, on the high side, because the pool's post-fee balance rounding is invisible from outside. That residual is why the fallback under-asks by 200 bps: the margin guarantees the pair's invariant check has slack at execution, converting a possible over-quote revert into a certain fill, and it is applied identically on the preview and in-frame surfaces so the two stay in lock-step. The canonical-oracle lane of §15 bounds the curve independently: against an implementation written from the venue's specification, the Core's stable quote sits within 4 wei over 5,000 fuzzed inputs.

§7 Deterministic Derivation $\mathcal{D}$

Before a pool can be quoted it must be found. The industry's answer is to ask: a call to the venue's factory reads a storage mapping and returns an address, and the caller takes the factory's word. BlazePhoenix supports that answer — four factory-call modes cover the venues whose deployments are not reproducible — and prefers a stronger one wherever the arithmetic allows: compute the address, with no external call. An address that is computed can be verified by anyone with the same three inputs; an address that is fetched can only be trusted.

The mechanism is CREATE2. The EVM makes a contract's address a pure function of its deployer, a salt and the hash of its init code:

$$\text{pool} \;=\; \text{addr}_{20}(\text{keccak256}(\texttt{0xff} \,\|\, \text{origin} \,\|\, \text{salt} \,\|\, \text{initCodeHash})) \tag{E8}$$

where addr_{20} takes the last twenty bytes of the hash. The Core spells the 85-byte preimage out exactly once (`create2Address`); `deriveAddress` selects the salt polynomial and the origin from the mode, and the Hub's attested Algebra derivation reuses the same producer.

MODE	FAMILY	RESOLUTION	SALT / PROBE DETAIL
0	V2	factory call `getPair(t0,t1)` — selector `0xe6a43905`	—
1	V3 / Algebra	factory call `getPool(t0,t1,fee)` — `0x1698ee82` ; on a miss, `poolByPair(t0,t1)` — `0xd9a641e1`	—
2	Solidly	factory call `getPool(t0,t1,stable)` — `0x79bc57d5`	—
3	Concentrated Solidly forks (Slipstream, Velodrome CL)	factory call `getPool(t0,t1,tickSpacing)` — `0x28af8d0b`	—
4	V2 CREATE2	pure arithmetic	`keccak256(packed(t0,t1))` , origin = factory

MODE	FAMILY	RESOLUTION	SALT / PROBE DETAIL
5	V3 CREATE2	pure arithmetic	`keccak256(encode(t0,t1,fee))` ; `fee == 0` is the Algebra sentinel → `keccak256(encode(t0,t1))` , origin = the factory's `poolDeployer()` (`0x3119049a`), attested at admission
6	Solidly clone CREATE2	pure arithmetic	`keccak256(packed(t0,t1,stable))` , origin = factory
7	CL CREATE2	pure arithmetic	`keccak256(encode(t0,t1,tickSpacing))` , origin = factory
8	*retired*	freed by the Curve excision; a tombstone — `MODES_VALID = 0x2FF` refuses it	—
9	V4 derive-scan	derive hookless poolIds and prove them live on the PoolManager (§5.1)	grid ≤ 99 tiers, ≤ 40 probed cold, stop at 8 pools

Modes 0–3 are one `staticcall` each; modes 4–7 are zero calls and require only the correct init-code hash; mode 9 serves the one family whose pools cannot be located either way.

A theorem, a claim, and a proof. A derived address is a theorem over the pair: given the origin, the salt polynomial and the init-code hash, the address follows by keccak alone, and anyone who disputes it can recompute it. An asked address is a factory's claim about its own storage — and a factory can answer with a pool on other tokens. Since 2026-09 an asked pool (modes 0–3) proves its own pair before discovery lists it: the Hub reads its `token0()` and `token1()` — the same two reads the Router makes at the seam that pays — and a mismatch drops the candidate, so a pool the executor would refuse is never listed, planned or ranked. The hostile-venue matrix of §15 is what changed this: its first run listed a pool a curator-admitted factory answered with, the planner ranked it, and the executor refused it at the seam that pays — funds never at risk, the pair refused while the impostor won the split.

Two subtleties in the V3-shaped rows. Algebra (Camelot-class) charges a dynamic fee, so a fee can never be part of its identity: an Algebra registration declares fee zero as a sentinel, mode 5 drops the fee from the salt, and the CREATE2 origin is the separate `poolDeployer` contract Algebra factories expose. That origin is the one derivation input a factory could still steer after admission — a proxy changes its answer with its codehash intact — so the Hub attests it once at `addFactory` (`factoryDeployer`) and derives from the attested origin thereafter; after control is renounced the attestation is frozen. On the factory-call side, Algebra factories expose `poolByPair` rather than `getPool` , so a mode-1 lookup that returns nothing retries under that selector before giving up.

Every derivation passes one final gate. The Hub discards any derived address that carries no runtime bytecode — `extcodesize` is the whole check (`hasCode`) — so a wrong init-code hash, a stale registration or a chain where a family was never deployed produces nothing rather than a phantom venue. Bytecode at the address proves existence, not honesty, which is why admission, the believability band and the floors still stand between a located pool and a filled trade. The same pool derived under several `(fee, spacing)` combinations is listed once.

The boundary between deriving and asking is empirical. Clone-based venues such as Velodrome and Aerodrome stay on factory-call even though their init-code hash is known, because their CREATE2 origin lives inside an internal deterministic-clone deployer rather than the factory — the address is not reproducible from public inputs, so the protocol asks. Uniswap V3 is the flagship case in the other direction: its init-code hash is one constant on every chain because the same pool bytecode is deployed everywhere, and the identity `derived == getPool` holds on forks of Ethereum, Arbitrum, Optimism and Base against each chain's own factories (§17). Derive where the arithmetic is sound, ask where it is not, and verify everything either way.

Discovery consumes the table row by row. For each of at most `MAX_FACTORIES = 16` registered factories the Hub probes every declared `(fee, spacing)` combination — Solidly rows in both stable and volatile flavours, a V4 row reserving up to eight — so the scan's worst case is fixed by the registration, not by the chain's mood. The configuration surface is guarded at the door (§9): a structurally impossible venue cannot be stored.

§8 The Monoslot

Storage is the dominant cost of doing anything on the EVM: a cold `SLOAD` is 2,100 gas, and a registry that scattered a pool's attributes across separate slots would pay it again for every attribute, for every pool, on every route evaluated. BlazePhoenix packs a pool's entire routing state into a single 256-bit word — the Monoslot — read once and decoded by pure arithmetic.

BITS	FIELD	WIDTH	MEANING
0	`active`	1	slot is live
1–7	reserved	7	bit 5 = stable curve (read from the pool's own `stable()` at registration); bit 6 = native side swapped; bit 7 formerly the bridge flag, no longer written — the bridge bonus is read live from the pair (§9)
8–31	`fee`	24	fee tier, in the venue's native unit
32–39	`kind`	8	venue kind (§4)
40–47	`tier`	8	0 = curated (seeded by an operator), 2 = permissionless (learned by trading or claimed)
48–59	`conc`	12	concentration field, masked to `0xFFF` before packing
60–63	`bucket`	4	logarithmic depth bucket
64–95	`lastUpdateTs`	32	wall-clock timestamp of the last touch
96–127	`emaIn`	32	reserved for an input-volume EMA; written as zero at registration, not updated by the shipped tick
128–159	`emaOut`	32	reserved for an output-volume EMA; as above
160–191	`swapCount`	32	activity count, saturating at $2^{32} - 1$

BITS	FIELD	WIDTH	MEANING
192–223	`regBlk`	32	registration block
224–255	`lastBlk`	32	last-touch block

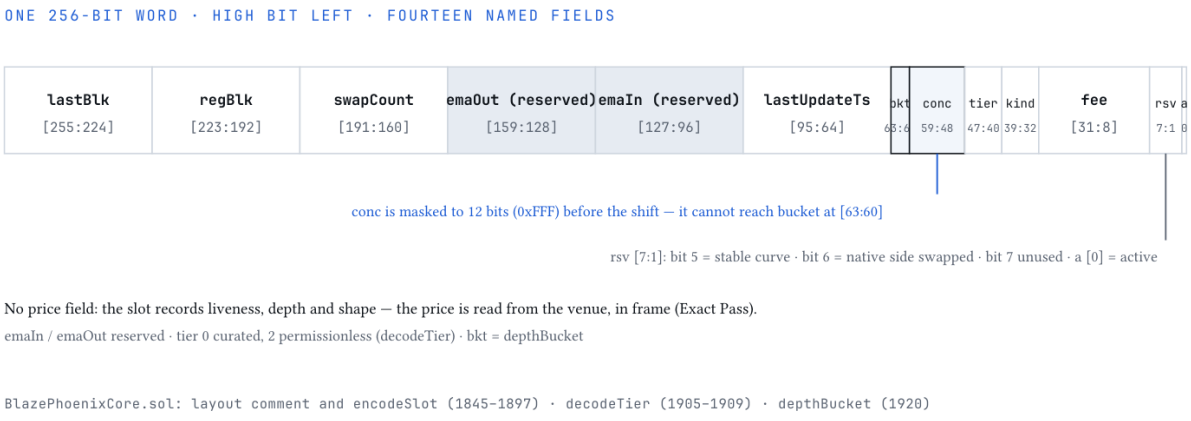


FIG. 4 Monoslot bit layout: one 256-bit word, fourteen named fields, the concentration field masked to twelve bits so it can never bleed into the depth bucket.

The encoding is a sequence of shifts and ORs (`encodeSlot`):

$$s = a \mid f \ll 8 \mid k \ll 32 \mid r \ll 40 \mid (c \wedge \texttt{0xFFF}) \ll 48 \mid b \ll 60 \mid t \ll 64 \mid e_{\text{in}} \ll 96 \mid e_{\text{out}} \ll 128 \mid n \ll 160 \mid B_{\text{reg}} \ll 192 \mid B_{\text{last}} \ll 22$$

(E9)

One detail is a guard, not a convenience: the concentration field is masked to twelve bits *before* shifting, so it can never bleed into the depth bucket at bits [63:60] — the packing forbids the class of silent field overlap that corrupts packed state, rather than trusting callers to stay in range. Decoding is equally mechanical: each accessor is a shift and a truncation; the bucket writer clears its four bits before setting them; the swap count saturates instead of wrapping, so a full slot can never roll a busy pool's history over to zero. The wall-clock timestamp, stamped by the Hub on every tick and registration, drives both the vitality decay of §9 and the Solver's discovery-freshness gate.

Notice what the word does not contain: a price. The Monoslot carries everything needed to rank a pool — activity, depth class, fee, kind, flags, recency — and nothing that could be mistaken for a quote, so the registry's memory is structurally incapable of pricing a trade. Prices are computed fresh, from live reserves, inside the frame that trades on them; the packed word only ever decides which pools are worth asking. The update path honours the same economy: a swap through a known pool rewrites the word once — count, bucket, timestamps, one `SSTORE` over one `SLOAD` — so the marginal cost of keeping the registry current rides inside transactions users were already sending.

Why one word matters is a question of multiplication. The Solver probes up to `MAX_CANDIDATES = 8` venues per pair across up to three topologies, so a single solved trade can score a couple of dozen pools before it moves a token. At 2,100 gas per cold slot, a five-slot-per-pool layout prices that scoring pass in the hundreds

of thousands of gas; the Monoslot prices it at one read per pool, and the Hub's `psisOf` returns every candidate's score in one batched call. The Monoslot is not an optimisation applied to on-chain routing; it is the reason on-chain routing is affordable at all.

§9 The Self-Healing Registry and the Vitality Field Ψ

Most aggregators know their venues because a keeper indexes pools off-chain and pushes lists on-chain — a trusted process, a staleness window and an operational dependency, three things a protocol that solves routes on-chain cannot accept. BlazePhoenix takes the opposite stance: the registry learns liquidity by trading it.

§9.1 Learning by trading, and the pair-proof

On every executed leg the Router calls the Hub's `recordSwap` (`onlyRouter`, `whenLive`), passing the pool, its kind, its fee, the pair, the amounts and the depth it measured. The Hub first refuses any kind the Router could not have executed (`KINDS_EXECUTABLE`) — a local defence that does not depend on the Router staying what it was at deploy — and then takes one of two paths.

A known pool's Monoslot is ticked: the swap count rises from its *currently decayed* base, the depth bucket is recomputed, the block and the wall-clock timestamp advance — one packed word rewritten, no external call. An unknown pool takes the cold path. It must first clear admission (`_canInsert`, below). Then it must prove the claim it is about to store: for every pair-shaped kind (`KINDS_PAIR_PROOF` = V2, V3, Solidly, Algebra) the Hub reads the pool's own `token0()` and `token1()` and requires them to match the pair — at most two `staticcall`s, on the cold path only. Every argument to `recordSwap` is caller-controlled calldata carried in the Router's route — pool, kind and depth — so without this proof an attacker registers a contract they wrote, under a pair they picked, at a depth they picked, holding neither token. For the two V4 kinds, whose `pool` field is a truncated id with no bytecode, the proof is re-derivation: the tick spacing is recovered and the id recomputed and matched, hookless-only, so a hooked pool can never register through this door; a native V4 pool is matched in both orientations of `(address(0), T)`. A mismatch *skips* registration and never reverts, because the user's swap has already settled and must never fail over a registry decision.

The pair-proof at registration is new in September 2026, and the measurement that produced it is §15's: fifteen source mutations were run against every stateful invariant, and the mutant that removed this proof survived every campaign on the first run, because no test had ever offered the Hub a pool trading other tokens. The Hub campaign now does so on every run — the pair reversed, one token shared, two foreign — and `invariant_ActiveEntriesTradeThePair` reads every active entry's `token0` / `token1`; the mutant is a guard entry that dies to it. Discovery carries the same proof for asked pools (§7).

There is no indexer to run and no moment at which the protocol is blind because a server is down. An operator `seedPool` door pre-populates pairs at deployment (tier 0, curated) and a permissionless `claimV4` door admits hookless V4 pools under their own proof (§5.1); both are optimisations, not dependencies — an empty registry heals itself into a populated one through ordinary trading.

§9.2 The score

The Vitality Field Ψ separates quality from capacity, reading only the Monoslot and the pair's live bridge status (Core `psi`):

$$\Psi(s,t) \;=\; V(s,t)\cdot 2^{\,b(s)}\cdot\left(1+\tfrac{2{,}500}{10^4}\,\mathbf{1}_{\text{bridge}}\right)\cdot\left(1+\tfrac{500}{10^4}\,\mathbf{1}_{\text{conc}}\right) \tag{E10}$$

The depth bucket $b=\min(15,\lfloor\log_{10}(d/10^{15})\rfloor)$ enters through its weight 2^b: each decade of depth doubles the weight, so depth counts sub-linearly — a deep pool is firmly favoured, but a merely healthy pool is not erased by a giant. A pool on a bridge pair earns +25 %, because bridges are the connective tissue of the liquidity graph; a concentrated pool earns a further +5 %. The bridge bonus is read live from the pair by one producer (`_pairBridged`) rather than from a bit frozen at registration, and within a pair it is uniform, so it can never reorder pools of the same pair.

The decay — one mechanism. Activity decays with idleness by a single wall-clock rule (`_decayedSwapCount`):

$$V(s,t) \;=\; \text{swapCount}(s) \;\gg\; \left\lfloor \frac{t-t_{\text{last}}}{24{,}576\ \text{s}} \right\rfloor, \qquad V=0 \text{ once the shift exceeds } 31. \tag{E11}$$

The step is `VITALITY_DECAY_STEP_SECONDS = 24,576` — about 6.8 hours — and thirty-two halvings extinguish any score, so a silent pool reaches zero and full evictability after 786,432 s, about 9.1 days. A single swap revives it, and revival is honest: the count resumes from what the decay left standing, never from the pre-decay total, so one dust swap after a long silence cannot restore a pool's accumulated rank. The scorer keeps two zeros apart: a *dead* slot — empty, never ticked, or past the horizon — scores a true zero and drops from ranking, while a *live* slot whose small count merely rounded away under the shift is floored to one, so a real, recently touched pool never vanishes on a rounding artefact. The window is in wall-clock seconds deliberately: a block-counted window means a different real duration on every chain, and an earlier implementation that shifted every 16 blocks decayed to zero about 128× faster than documented.

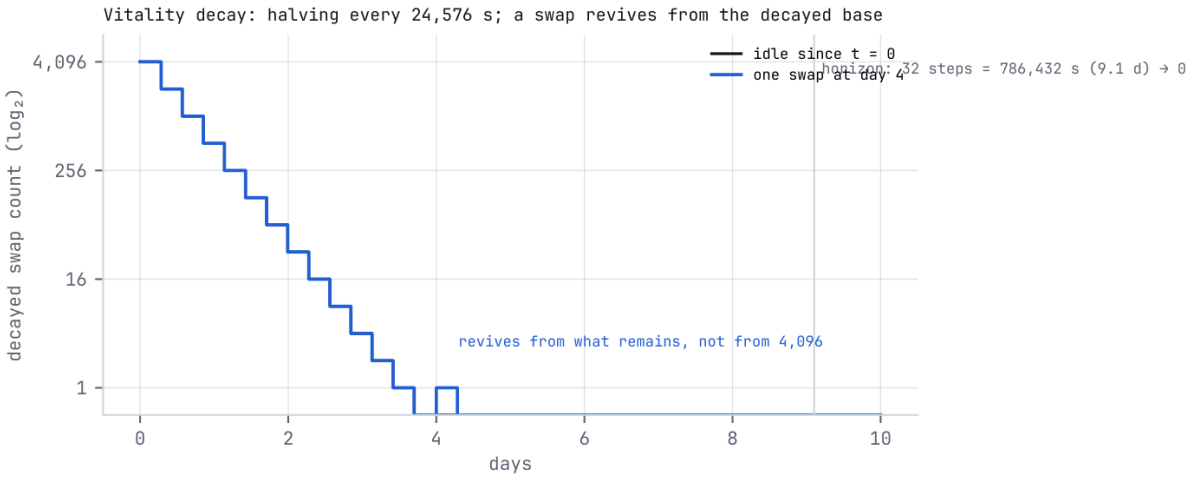


FIG. 5 Vitality decay: a count of 4,096 halves every 24,576 s and reaches zero after 32 steps; a swap at day 4 revives from the decayed base, not from the historical total.

§9.3 Eviction

Each pair holds at most `MAX_SLOTS = 16` pools. When a seventeenth wants in, the Hub scores every incumbent by full Ψ — depth, bonuses and all, so a deep newcomer can displace a shallow-but-warm squatter — and admits the newcomer only if its projected score clears the weakest incumbent by a margin (`_canInsert`):

$$\Psi_{\text{new}} \;>\; \Psi_{\text{worst}} + \lfloor \Psi_{\text{worst}}/4 \rfloor, \qquad \Psi_{\text{new}} = 2^{\,b(d_{\text{new}})} \tag{E13}$$

with the newcomer scored conservatively — vitality one, its own depth bucket, no bonus — and a zero-depth newcomer refused outright. The margin closes a griefing vector: under a bare displace-the-lowest rule, an adversary could hold sixteen shallow slots barely active and keep a deep competitor out; ranked by Ψ with a margin, the genuinely deeper pool still wins. The constant's limit is stated where it is enforced: the newcomer's weight is a power of two while the incumbent's score is continuous, so the effective margin runs from about 100 % (an incumbent just above one) down to about 25 % (an incumbent at 51, smallest admissible newcomer 64); the rule's 25 % is also the floor of the behaviour, never undercut. A heavily traded incumbent is hard to displace until decay does its work — two idle days cost it seven halvings. A permissionless V4 claim clears the same margin with the pool's measured liquidity as its depth.

§9.4 Discovery, and its gas knob

Discovery is a public view: for any pair the Hub walks its factory list, derives candidates by the modes of §7, applies `hasCode` and the pair-proof, and the Solver merges the result with the registered set, deduplicated by address, so neither source can suppress the other. To bound gas, the Solver applies a freshness gate (`_registryFresh`): when at least `MIN_FRESH_VENUES = 3` *curated* registered venues for the pair have been touched within `DISCOVERY_TTL_SECONDS = 3,600` , it trusts the registry and skips the derivation sweep. Only curated rows (tier 0) can vouch: a permissionless row is written by whoever swapped first, and counting it let three self-registered dust pairs, kept fresh by three dust swaps an hour, switch discovery off for a pair and hide every honest venue never registered. The permissionless row stays a candidate; it no longer chooses the path. The gate is a gas and coverage knob and never a safety parameter — the Iron-Law floor and the user's minimum protect every fill regardless of how stale the registry is, because neither reads the registry at all.

§9.5 The configuration surface

The Hub's registration doors are guarded on the way in. `addFactory` rejects, before any state is written (`HubE(5)`): a kind outside `KINDS_ROUTABLE` ; a mode outside `MODES_VALID` ; a CREATE2 mode with a zero init-code hash; a V2-salt slot bound to anything but the V2 kind; a clone slot bound to a non-Solidly kind; a V3-salt slot bound to anything but V3 or Algebra, with an Algebra registration required to declare every fee as the zero sentinel; and a V4 derive-scan row bound to any kind but V4 or with mispaired fee and spacing lists. A known factory is refreshed in place, never given a second row, so one address cannot exhaust the sixteen seats (`HubE(4)`); after control is renounced, a factory whose runtime code moved cannot be re-armed, and a derive row cannot be converted into an ask row. `allowHook` pins the hook's codehash at admission, and a hook is routable only while allow-listed *and* unchanged (`isHookLive`); after renunciation, de-listing is gone and a moved hook stays paused for ever. Bridges are at most `MAX_BRIDGES = 3` ; `addBridge` is a curator power that survives renunciation, `removeBridge` a control power that dies with it.

§9.6 The structural boundary

One sentence bounds everything above: **Ψ can hide a venue; it can never misprice one.** The score ranks and truncates the candidate set — no quote, no floor and no allocation reads it, because those are computed from in-transaction measurements; memory reaches route selection only by omission. Whatever the registry believes, the worst it can surface is a worse-but-real live price, and the value such a price can pass through is capped by the strictest of the believability band, the Iron-Law floor re-derived in-frame, and the minimum the user set. Principal is out of reach by construction. The registry decides what is seen; it has no vote on what is paid.

§10 Routing and the Split Allocator

A constant-product pool prices by one rule: its two reserves, multiplied, may never shrink. Pour Δ in with fee retention γ (a 0.30 % venue fee means $\gamma = 0.997$) and you receive $y\gamma\Delta/(x+\gamma\Delta)$, with your own input in the denominator: the more you push through one pool, the worse each further token pays. Slippage is arithmetic, and so is its cure: splitting the same order across pools of the same pair moves each pool less.

Worked example 2 — a split across two pools. One order of 100 against two pools of the same pair, both at 0.30 %: pool A holds 1,000 a side, pool B holds 4,000 a side.

ROUTE	ARITHMETIC	YOU RECEIVE
All 100 into A	$1{,}000 \times 99.7/1{,}099.7$	90.661
All 100 into B	$4{,}000 \times 99.7/4{,}099.7$	97.275
20 into A, 80 into B	$1{,}000 \times 19.94/1{,}019.94 \;+\; 4{,}000 \times 79.76/4{,}079.76$	$19.550 + 78.201 = 97.751$

The 20 / 80 split is the allocator's own choice, not a guess: weights are proportional to measured depth against the deepest candidate — A weighs $1{,}000/4{,}000 = 2{,}500$ of 10,000, B weighs 10,000 — so A takes $2{,}500/12{,}500 = 20$ % of the order. The split beats the best single pool by 0.476 tokens, 48.9 bps of it. The split gate then asks whether that gain earns its second leg: the threshold is $97.275 \times (1 + 25/10^6) = 97.278$, and 97.751 clears it, so the split stands. Every row is checkable with a calculator, and it compounds: price impact is each pool's response to *its own* slice, so every admissible leg added to a route moves every pool less.

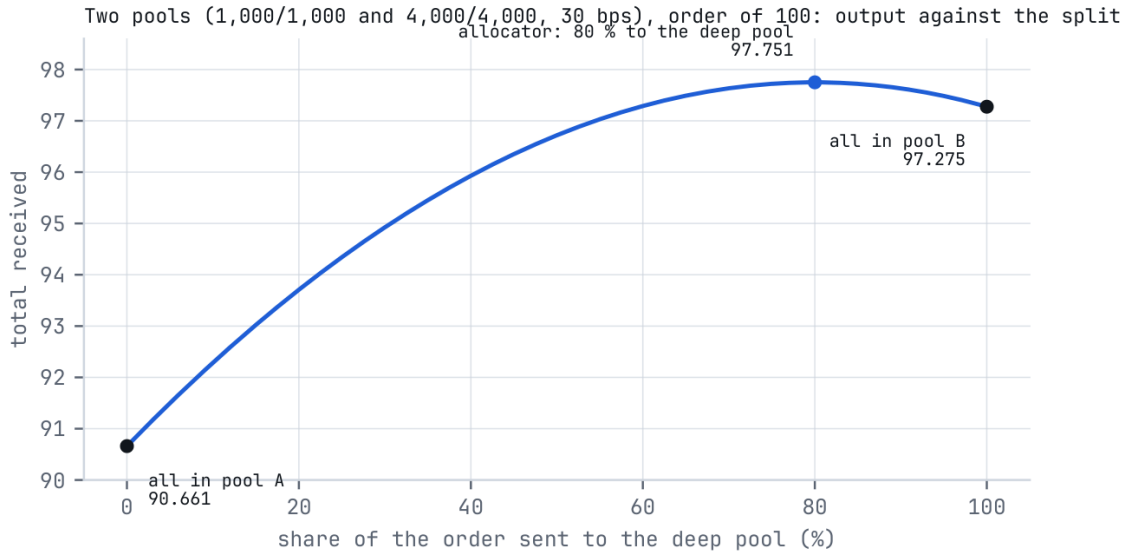


FIG. 6 Output against the share sent to the deep pool for the two-pool example: a concave curve with its maximum near 80 %, the allocator's depth-proportional point marked.

§10.1 Route shapes and budgets

The Solver builds three topologies and returns the maximiser (`findBestRoutePlan` , with the runner-up as a fallback): a *direct* route in one hop; a *bridge* route through one intermediate; a *double-bridge* route through two — three hops, the deepest shape the Solver builds and the most the Router accepts (`MAX_HOPS = 3`). Every hop splits across parallel legs under two deliberately separate budgets: `MAX_LEGS_PER_STAGE = 4` answers *how many legs fit in a hop*, `MAX_LEGS = 11` answers *how many fit in a route*, and together they yield 4, 4 + 4 and 4 + 4 + 3 — the last hop of a three-hop route gets three because the global ceiling squeezes it, not because anyone wrote a special case. Bridges come from an operator-configured set of at most three tokens in the Hub; the Solver expands every configured bridge and lets the measured `totalOut` decide, per trade. A route whose measured impact exceeds `MAX_ROUTE_IMPACT_BPS = 9,000` is not returned at all, and a pair with no admissible route refuses with `SolverE(5)` rather than fabricating one. The Router enforces its own five-leg cap per hop on any caller-supplied route (`MAX_LEGS_PER_HOP = 5` , `RouterE(3)`).

§10.2 The funnel

For each pair the Solver gathers the registered pools and, unless the registry is fresh (§9.4), the discovered ones; removes any pool whose hook alters deltas (the projector Ξ, applied here so that the planned door can never assemble a route the Router would refuse); ranks the union by Ψ in one batched call and keeps `MAX_CANDIDATES = 8` — deliberately wider than any hop's leg budget, so a deep pool listed behind several thin venues is seen, weighted and can displace them instead of being starved by list order. An exact tie in Ψ breaks toward the lower fee, so on a cold registry the funnel is a function of price rather than of history.

Each candidate is then probed once, at a hundredth of the order (`amountIn / 100` , or the whole order when that floors to zero), capturing its marginal rate and its depth in the same call. Marginal rates cleanly separate two signals that full-size rates conflate: a stale-priced pool has a wrong marginal rate regardless of size, while

a small but healthy pool has a correct marginal rate and merely a poor full-size output. For pair-shaped kinds the declared depth is capped by what the pool physically holds of the output token, so a synthetic pair declaring 3×10^{30} while holding one token of each side is weighed at what it holds.

§10.3 The believability band

Before allocation, candidates must be believed, and belief must be anchored on a quantity that costs something to forge. The band's centre is the **depth-weighted median** of the survivors' marginal rates (`_depthWeightedMedian`): rates are sorted with their depths travelling alongside, and the median is the rate at which half the *depth mass* has been passed. Its breakdown point is 50 % of the mass — to move it, an attacker must out-depth half the pair's real liquidity — and the depths it weighs are `getReserves` for reserve pools, active liquidity for concentrated pools and measured liquidity for V4: quantities a plain donation to a pool does not move. That last property is why the anchor is not the largest real balance: a transfer to a pool inflates `balanceOf` without moving the reserve or the price and is recoverable by the donor, so it is not capital at risk — an external researcher demonstrated the earlier balance anchor being captured by a donation, and the anchor was replaced. Candidates whose marginal rate departs from the centre by more than `MEDIAN_FILTER_BPS = 500` (±5 %) are dropped before anything is sent; a candidate set of one is not filtered at all, and a set whose centre rounds to zero falls back to the plain median so an otherwise routable hop is not killed by a rounding artefact. The width is a calibrated choice: Solidly stable curves quoting liquid-staking pairs typically sit 20–40 % from the true rate and are excluded; a healthy pool whose reading for this trade size differs a little from the centre is admitted. The metamorphic lane of §15 measured the band from outside: with one pool priced 10^{10} away from its sibling, the plan pays 35 % less than the outlier alone — the median refusing to believe it.

§10.4 The allocator and its three guards

Weights are proportional to measured depth, normalised against the largest depth in the set (`_weights`) — one normalisation, no modes, because every family's depth is token-denominated (§4.2) and therefore comparable. Proportional allocation is the robust choice for an adversarial environment: it needs no iteration and no per-venue curvature model, and it avoids the catastrophic failure a uniform split produces — in a measured WETH/USDC case, splitting 50 / 50 across a 790-WETH pool and a 0.34-WETH dust pool returned 1,302 USDC where concentrating in the deep pool returned 1,638; the depth-weighted allocation returned 1,638. When the funnel holds more survivors than the hop's budget, the top-budget survivors *by weight* are kept, so a deep late-listed pool displaces thin early-listed ones. Three guards then stand between the allocation and the route it becomes.

- **The capacity clamp.** The single-tick concentrated-liquidity formula models the current tick's liquidity as if it spanned every price, so on a thin pool it can promise more than the pool has ever held — 117× more in one observed case. A concentrated leg on a pool-shaped venue (kinds 1 and 6) is therefore clamped to `MAX_CONC_DRAIN_BPS = 3,000` — 30 % — of the pool's *measured* real output-token balance. The clamp does not reach V4 legs: the singleton holds every pool's tokens in one balance, so the Solver skips the read for kinds 4 and 8 and a V4 promise carries no physical bound beyond the floor of §11.3 — the floor is its bound. The clamp is two-tier: when the quote exceeds the pool's entire holdings the promise is physically impossible, and the clamp cuts the leg's *committed input* in the same ratio, cascading the freed input to the remaining legs and leaving anything unroutable to be swept back to the caller; when the fill is aggressive but possible, only the promise is capped, because

cutting capital there was measured to force partial fills on pools that execute fine. Reserve-bounded formulas cannot over-promise and are untouched. This clamp reads `balanceOf` deliberately where the band does not: the band asks a *relative* question a donation can skew, the clamp asks a *physical* one — can this pool pay this? — and a donation raises what the pool can really pay.

- **The split gate.** Each extra leg costs real gas, and the allocator deliberately contains no gas-price term, because comparing gas in the native token against output in the trade's token would need exactly the price oracle the protocol refuses to have. The oracle-free proxy is a threshold: a split is kept only if $\text{out}_{\text{split}} \geq \text{out}_{\text{single}} \cdot (1 + 25/10^6)$ — `MIN_SPLIT_IMPROVEMENT_PPM = 25` — otherwise the allocation collapses to one leg. The threshold is a measured number: the true break-even between a leg's gas and its improvement measured 3 ppm on Base (320 bytes of calldata per extra leg at 105.5 gas per byte), and the gate sits at about eight times that, denominated in parts per million because a quarter of a basis point does not exist in basis points; its history — 20 bps, then 5, then 0.25 — is a record of two values that had never been calibrated against a measurement. When the gate collapses a route, it measures two single-leg candidates at full size rather than one — the deepest pool and the best marginal rate — because marginal rates flatter shallow pools while depth is the proxy for carrying the whole size; the better fill wins.
- **Insurance by measurement, not by margin.** An earlier design shaved every leg of longer routes by 5 bps at planning time as insurance against inter-leg drift. The drift was then measured on nine Base pairs in the same block: worst case −2 bps, an order of magnitude below the insurance, which every honest multi-leg route paid on every trade — 27× its measured hazard. The shave was retired; the floors, enforced on realised output, carry that risk at zero planning cost. The measurement was taken in a calm window and the counter-argument is on the record in the source: if conservatism is ever restored, the right place is the floor, which is checked, and not the estimate, which only ranks.

The assembled route carries what the Router will re-derive: each leg's committed input and expected output, the route's share-weighted impact, and the floor Φ the Solver attests from the same arithmetic the Router enforces (§11). The Solver also prices each leg's execution by kind from the `THETA_GAS` ladder and publishes the total as `estGas`, an advisory figure the route carries and nothing on-chain reads. The reader's rule of thumb survives all of this machinery unchanged: compare the improvement the router claims against the gas your wallet shows, before you confirm.

§11 The execution layer and the Iron-Law floor Φ

The Solver proposes; the Router disposes — atomically, or not at all. The Router is the only contract that moves funds, holds none between transactions, and is built on one principle: one mechanism per concern, applied universally. It assumes nothing it can measure.

§11.1 Four doors, one executor

DOOR	SIGNATURE (ABRIDGED)	AUTHORISATION	ROUTE
`swapExactIn`	`(route, amountIn, userMinOut, recipient, deadline)`	classic `approve` + `transferFrom`	from calldata

DOOR	SIGNATURE (ABRIDGED)	AUTHORISATION	ROUTE
`swapExactInWithPermit2`	`(route, amountIn, userMinOut, recipient, deadline, permit, signature)`	one-step Permit2 signature transfer through the canonical Permit2 contract; no standing allowance to the Router	from calldata
`swapExactInNative`	`(route, userMinOut, recipient, deadline)` payable	native value, wrapped once at the door; fail-closed until `weth` is wired (`RouterE(3)`)	from calldata
`swapBestExactIn`	`(tokenIn, tokenOut, amountIn, userMinOut, recipient, deadline)`	classic	solved in-frame by the Solver (`findBestRoutePlan`), then executed through a self-call with the payer threaded explicitly

Only `swapBestExactIn` invokes the Solver in the transaction; the other three take the route the caller computed off-chain, for free, through the Quoter (§12). All four converge on one private executor (`_execute`), so every check below runs on every door. An EIP-7702 delegated account calls the same doors as a contract would; no separate entry exists for it.

Two refusals run before anything moves. A zero minimum is refused at every door — `RouterE(10)` , no exceptions, no compatibility flag — because the minimum is the one bound the contract cannot compute on the user's behalf, and a guard that may be absent is not a guard. And a route with more than three hops, more than five legs in a hop, or a hop whose input token is not the previous hop's output is refused with `RouterE(3)` : hop continuity is what makes the pre-swap "foreign" balance of every intermediate token known, so a crafted route cannot name a stranded token as its input and scale the Router's whole balance of it into the swap.

§11.2 Measurement, from the first instruction

Whichever door is used, the first thing the Router does is measure what arrived: it reads its own balance delta after the pull rather than trusting the nominal amount, so a token that delivers short on transfer enters the route at its true size. Baselines are born once: the input token's pre-swap balance (`baseIn`) and the output token's (`toutStart`) are taken at entry, and every later measurement is a delta against them, so tokens already sitting in the contract — dust, donations, strandings from unrelated history — are excluded from what a swap can pay out. A route that manages to spend below its own baseline fails on checked arithmetic — the safe outcome, since those funds were never this swap's to spend.

A multi-hop route is a plan, and plans meet reality at every hop boundary. The Router rescales each hop's legs against the balance that *actually* arrived — hop 0 against the measured post-pull input, capped by what the route committed (so the Router never force-feeds a pre-cut order into a thin pool); every later hop against the measured bridge balance above its foreign baseline — each leg executing $\text{amt} = \text{amountIn}_{\text{leg}} \cdot \text{real}/\text{planned}$ (`_hopScaleImpactAndQuote`). A leg scaled to zero is skipped; unspent input and bridge residuals are swept back to the payer, never to the Router. The same pass reads each leg's pool state once and produces, from that single read, the leg's scaled amount, its measured impact and its in-frame quote — the three quantities the floors consume — so the reserve and concentrated families pay no extra `staticcall` for their measurement.

§11.3 Three floors under every leg

The per-leg floor. Each leg's measured contribution to the Router's output balance must reach `LEG_FLOOR_BPS = 8,000` — 80 % — of its bound, or the whole swap reverts with `RouterE(5)` (`_execScaled`), so a single sandwiched or manipulated pool fails the transaction immediately instead of hiding its loss inside an otherwise healthy total. The bound is the caller's attested quote, pro-rata to the amount actually spent — *lifted* to the quote measured in-frame whenever the attestation covers less than `MIN_QUOTE_COVERAGE_BPS = 5,000` of it. That is a maximum, not a minimum, on purpose: on a floor, `min(claimed, measured)` with a deflated claim returns the deflated one and relaxes, which is the attack the gate closes. The guard runs whenever there is any floor basis — the caller's attestation or the in-frame quote — so writing zero into a route no longer switches the caller's own floor off. Its limit: a venue with no measurable quote and no attestation falls through to the aggregate floors and the user minimum, which bound it anyway.

The aggregate floor per hop. The per-leg floor is local while composition is global: an attacker holding one leg of L could extract about $20\,\% \cdot (L-1)/L$ of a hop without failing any single floor. So each hop may lose at most what one *average* attested leg could legitimately lose: $\sum \text{got} + \lfloor (\text{BPS} - \text{LEG_FLOOR_BPS}) \cdot \lfloor \sum \text{attested}/n \rfloor / \text{BPS} \rfloor \geq \sum \text{attested}$ — the slack is 20 % of the average attested leg, else `RouterE(5)`. The mean and not the maximum, because under a maximum the attacker inflates their own leg to inflate the shared budget. For one leg the rule collapses exactly onto the per-leg floor — zero new rigidity, by construction.

The route floor Φ is §11.5.

§11.4 One callback, one unlock

Every V3-shaped venue — Uniswap V3, Algebra, the concentrated Solidly forks, Pancake V3, Sushi V3 — uses the same flash-accounting pattern under a different callback name: the pool calls back mid-swap with a delta for token0, a delta for token1 and arbitrary data, demanding payment. Rather than one named callback per fork, the Router exposes a single `fallback` that answers them all — the **One-Callback Doctrine**: it reads the two deltas, identifies which token it owes from their signs, and pays exactly that amount, taking the in-flight leg context from transient storage (`TSLOT_POOL`, `TSLOT_TOKEN`, `TSLOT_AMT`). Because a fallback is reachable by anyone, 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 — a second demand within one leg is refused with `RouterE(6)`. A newly launched V3 fork with a novel callback name is supported the day it deploys.

The V4 singleton is wrapped behind the same hop interface: `unlock`, then swap, `sync → settle → take`, re-lock, with the currencies carried in transient storage and the native seam of §5.3. Before any V4 leg executes, the Router reads the hook address and refuses the leg if its bits declare delta-altering permissions (`RouterE(9)`); the sieved hook is the executed hook, because the canonical PoolManager echoes the unlock data verbatim (`test/V4SievedHookIsTheExecutedHook.t.sol`). The route-shape rule is canonical order, route-wide: once a hooked leg has executed, no hookless leg executes after it in this or any later hop (`sawHooked`), so a hook cannot observe a later hookless leg of the same route before it settles; a hooked leg may sit in an earlier hop only if every leg after it is hooked too.

Fee-on-transfer, on the legs that can afford it. Some tokens deliver short on every transfer. On constant-product legs the Router handles this by the canonical pattern — transfer first, then read the pool's reserves and its balance in the same post-transfer state, and price the output on the difference the pool actually sees — a quantity correct by construction, because it is exactly the quantity the pool's own invariant will check. The branch engages only when a token measures short (`TSLOT_FOT`), and what engaging it costs is stated with it: the quote-derived route floor was computed blind to the tax and is dropped for that swap; the protocol floor is re-scaled by the measured net ratio (rounded up, the conservative direction); the per-leg bounds are re-priced by the same ratio; and the user's minimum becomes the binding constraint. Concentrated legs do not support fee-on-transfer by design — flash accounting must pay exactly what is owed — so a taxed token on a V3-only route is refused up front with `RouterE(13)` rather than reverting inside a pool.

§11.5 The Iron-Law floor Φ, in full

Not every route that can execute should — a route returning 60 % of fair value is worse than no trade — so under everything above sits a floor the protocol derives for itself. It is a retention fraction in basis points (Core `ironFloorBpsShv` , `legShaveBps`):

$$\texttt{floorBps} \;=\; \max\!\big(\; 9{,}600 - \texttt{legShv} - \min(\delta,\, 10^{4}) - \sigma_{\ln}/10^{14},\;\; 8{,}000 \;\big) \tag{E15}$$

$$\texttt{legShv} \;=\; \sum_{\text{hops}} \left\lfloor\, 200 \cdot \frac{(\sum_i a_i)^2 - \sum_i a_i^2}{\sum_i a_i^2} \,\right\rfloor \;=\; \sum_{\text{hops}} \lfloor 200\,(n_{\text{eff}} - 1) \rfloor \tag{E23}$$

Three constants decide it: `FLOOR_BASE_BPS = 9,600`, `FLOOR_PER_LEG_BPS = 200`, `FLOOR_HARD_MAX_LOSS_BPS = 2,000`. The floor starts at 96 % for a clean single-leg swap, loosens one basis point per basis point of measured impact δ and 200 bps per *effective* extra leg, and is clamped so it never falls below 8,000 bps: the protocol will never knowingly admit a route retaining less than 80 % of its reference, whatever the inputs. The leg shave is computed from the *concentration* of the trade across legs — $(\sum a)^2 / \sum a^2$, the effective number of legs, exactly N for N equal shares and exactly 1 for one dominant leg — so a split earns loosening in proportion to how genuinely it is split, and a leg carrying a vanishing share earns a vanishing amount whether it declares zero or one wei. By Cauchy–Schwarz the numerator never underflows. The impact term is the share-weighted mean of the legs' measured impacts over the route. Two properties are printed as part of the definition: the volatility term $\sigma_{\ln}$ exists in the signature and is passed as literal zero at every call site — a hook for a future estimator, currently inert — and the reserve-family impact is fee-exclusive while the concentrated-family impact is fee-inclusive, each in the form its family's specification gives it.

What makes Φ an enforcement rather than a heuristic is where it runs and on what. The route handed to the Router carries advisory fields — a quoted total, an attested floor — and the Router trusts neither: inside the executing frame it measures each leg's real impact, computes the leg shave from the amounts it is handed, recomputes `floorBps` , and applies it to the **final hop's in-frame quote at the amount that actually arrived there** — a reference denominated in the output token, derived from pool state read during this execution, unforgeable by calldata. The choice of reference is what lets the guard fire at all: a fraction of the *realised* output can never exceed the realised output. When the final hop cannot be quoted in-frame — a liquidity gap at the current tick, a fee sentinel — the reference falls back to the hop's attested quote, which the coverage gate has already lifted and the fee-on-transfer measurement re-priced, so a just-in-time (JIT) liquidity gap

on a thin pool cannot disarm the floor; the floor's anchor is always a hop that moved value and produces the route's output. The check runs on the gross output, before any output-side fee. The effective minimum is a three-way maximum,

$$\texttt{effMin} \;=\; \max\!\left(\texttt{userMinOut},\;\; \texttt{route.singleOutFloor},\;\; \left\lceil \texttt{finalHopQuote} \cdot \texttt{floorBps} / 10^{4} \right\rceil\right) \tag{E16}$$

and its asymmetry is the point: caller-supplied fields can tighten protection and can never loosen it. The Solver attests `singleOutFloor` from the same `legShaveBps` and `ironFloorBpsShv` on the same amounts, so for a Solver-built route the attested floor and the enforced floor agree by construction (`test/FloorParitySolverRouter.t.sol`), and the regime harness of §15 pins the two frames on every generated row: with the fee off the input, the enforced floor sits in $[\text{attested} \times (1 - \text{fee}),\ \text{attested}]$; with the fee off the output, it equals the attested floor.

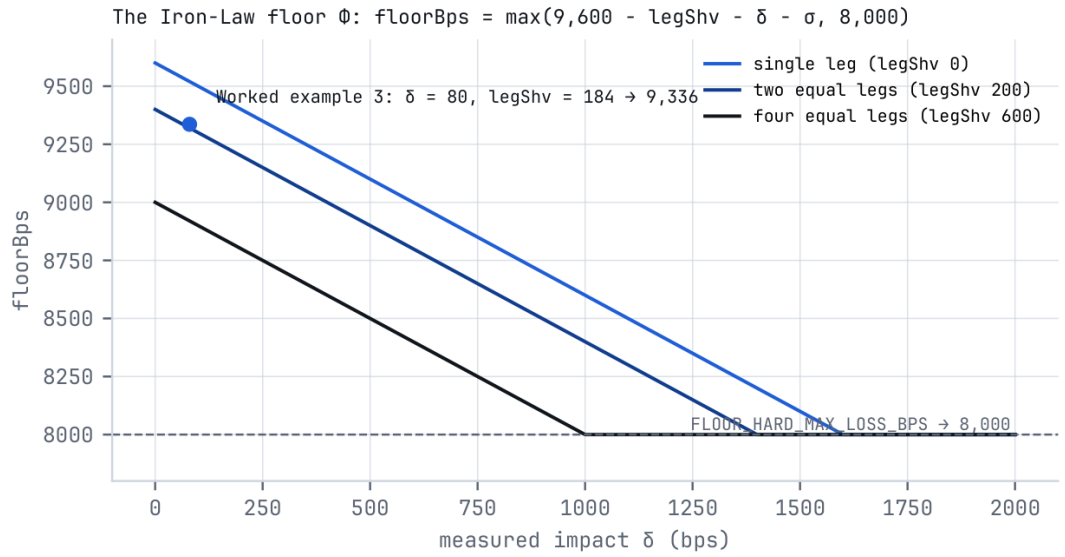


FIG. 7 The Iron-Law floor: retention against measured impact for a single leg, two equal legs and four equal legs; the 80 % hard clamp; the point of Worked example 3 marked.

Worked example 3 — the floor on a two-hop route. A route A → WETH → B: hop 0 splits 600 A into pool P and 400 A into pool Q; hop 1 sends the WETH produced through one pool R. Measured in the frame: hop 0's legs show impacts of 100 and 60 bps, hop 1's leg 40 bps, and the share-weighted mean over the route is 80 bps. Hop 1's in-frame quote at the WETH amount that actually arrived is 1,000.000 B.

1. Hop 0 concentration: $(600 + 400)^2 = 1{,}000{,}000$; $600^2 + 400^2 = 520{,}000$; $n_{\text{eff}} = 1.923$; shave $= \lfloor 200 \times 480{,}000 / 520{,}000 \rfloor = \lfloor 184.6 \rfloor = 184$ bps.
2. Hop 1 has one leg: $(\sum a)^2 - \sum a^2 = 0$; shave 0. Total `legShv` = 184.
3. `floorBps` $= \max(9{,}600 - 184 - 80 - 0,\ 8{,}000) = 9{,}336$: the route must retain 93.36 % of its reference.
4. `protocolFloorOut` $= \lceil 1{,}000.000 \times 9{,}336 / 10{,}000 \rceil = 933.600$ B.
5. With `userMinOut` = 990 B set by the trader and `singleOutFloor` = 933.6 B attested by the Solver, `effMin` = 990 B — the user's own bound is the binding one, as it should be on an honest route. A de-

livery of 998.4 B settles; a delivery of 985 B is refused with `RouterE(5)`, and the whole transaction, every pool's swap included, never happened.

Had the trader set `userMinOut` = 900 B, the protocol floor would bind at 933.6 B: a fill at 930 B, 7 % below the in-frame reference, is refused even though it clears the user's number.

§11.6 Settlement, and how to read one

Settlement closes the frame the way it opened: by measurement. The Router takes its output-token delta against the entry baseline as the gross output (`RouterE(8)` if it is zero), runs the floor check above, charges the fee where the regime puts it (§13), transfers the net to the recipient and measures the *recipient's* balance delta, enforces `userMinOut` on that delivered amount — so a fee-on-transfer output token cannot slip the user below their bound — and reads the fee ledger once: a settlement that paid the protocol nothing is refused (`RouterE(15)`), and on an anchored route a second payment is refused (`RouterE(16)`). Then, and only on the success path, it calls the Hub's `recordSwap` once per executed leg (`_recordHits`) with the amounts the hop was able to spend — the measured-over-declared ratio applied to each leg — which is the single external write a swap performs; nothing is written when the floor rejects.

Every settlement emits `ExecutionProof(user, tokenOut, quoted, realized, floorUsed, blockNumber)`: the reference quote produced in the same frame as the execution, published as an on-chain series that anyone can re-derive by an `eth_call` to `findBestRoutePlan` at that block. Its fields, without over-claiming: `user` is the payer, never `msg.sender`; `quoted` is the final hop's quote — it proves the last hop, not the route; `realized` is what was delivered, measured at the recipient; `floorUsed` is the protocol floor that had to be beaten. Each `Fee(token, amount, toT1, toT2)` event names the token and the split.

Worked example 6 — reading a settlement. The route of Example 3 settles and emits `ExecutionProof(0xUSER, B, 1,000.000, 998.400, 933.600, N)` and one `Fee(WETH, 0.014, 0.0042, 0.0098)`.

QUANTITY	VALUE	READING
`quoted`	1,000.000 B	the final hop's in-frame quote at the WETH that actually arrived
`realized`	998.400 B	delivered to the recipient, measured there; 9,984 bps of the quote
`floorUsed`	933.600 B	`floorBps` 9,336 applied to the quote; the fill sat 64.8 B above it
`Fee`	0.014 WETH in one event	the anchored regime: 28 bps of the 5.000 WETH that hop 0 produced, paid at hop 1's input, 30 % to the first treasury; nothing was taken from B
distance to `userMinOut`	8.4 B	the user's 990 B bound left 0.84 % of slack

Delivered minus quoted is the market's answer, not the protocol's: the fee left earlier, in WETH, and no output-side cut exists on this route. A `Fee` event denominated in the output token appears only on a direct route into a bridge coin (§13).

§11.7 The Router's refusals

CODE	MEANING	WHERE
`RouterE(1)`	unauthorised control call	`onlyControl`
`RouterE(2)`	paused, or renunciation attempted while paused	`whenLive` , `renounceControl`
`RouterE(3)`	bad input — zero address, unwired native door, hop count, legs per hop, hop discontinuity	entry points, `_execute`
`RouterE(4)`	deadline passed	entry points
`RouterE(5)`	output below a floor — per-leg, per-hop aggregate, route floor or `userMinOut`	`_execScaled` , `_execute`
`RouterE(6)`	callback not authorised — wrong pool, no leg in flight, second demand	`fallback`
`RouterE(7)`	re-entrancy	`nrEntrant` (transient lock)
`RouterE(8)`	swap failed — zero output, unknown kind, fee at or above the amount	`_execute` , `_execScaled` , `_chargeHopFee`
`RouterE(9)`	V4 hook alters deltas	V4 leg pre-commitment
`RouterE(10)`	`userMinOut == 0` with `amountIn > 0`	every door
`RouterE(13)`	fee-on-transfer token on a concentrated-only route	`_execute`
`RouterE(14)`	rescue not queued or inside its 48-hour delay	`executeRescue`
`RouterE(15)`	settled without paying the protocol fee	fee ledger, end of `_execute`
`RouterE(16)`	fee paid twice on an anchored route	fee ledger, end of `_execute`

Codes 11 and 12 are not assigned. A revert surface as enumerable as the API is what lets the regime arrays of §15 judge every row by one rule — settle, or refuse with a selector of ours — and call anything else a third way.

§12 Quote surfaces, and how a quote ages

There are two places a number called "your quote" can come from, and they are different code paths: the **Quoter**, a read-only preview contract an interface calls before you sign, and the **in-frame path**, where the Solver and Router compute quotes inside the executing transaction. §4.4 gave the per-kind truth; this section gives the surfaces and measures the gap between them.

§12.1 The Quoter's surfaces

FUNCTION	RETURNS	NOTES
`previewPlan(tIn, tOut, amountIn)`	a `Preview`	the Solver's plan, packed with the fee and the safety buffer
`previewPlanWithMinOut(…, userMinOut)`	a `Preview`	the same, with `canExecute` judged against the caller's minimum
`previewAndEncode(…)` / `previewAndEncodeWithMinOut(…)`	a `Preview` and the calldata for `swapExactIn`	the way an integrator takes a quote — the §12.2 study uses exactly this
`batchQuote(entries[])`	up to `MAX_BATCH = 32` previews	a pair that cannot be planned leaves a zero-initialised entry
`previewPlanExact(tIn, tOut, amountIn)`	the route and an exact net output	the Exact Pass: every concentrated leg dry-run against the pool's own swap

What a preview is. The `Preview` struct is advisory by design — the binding checks are the ones the Router re-derives in-frame — and its fields are documented at the definition site: `grossOut` echoes the plan's total; `protocolFee` is the effect of the fee on the output, modelled per regime exactly as the Router charges it (§13) — once on an anchored route, once per hop on an exhausted one, rounded up as the Router rounds; `safetyBuffer` is the inter-leg buffer

$$s \;=\; \min(\max(0,\, n-2) + 5a,\;\; 10)\ \text{bps} \tag{E21}$$

zero at two legs or fewer, one basis point per leg above two, plus five for each constant-product leg priced on an *assumed* fee (a declared fee the Core replaced under its ceiling), under one shared cap of ten, because a buffer without a ceiling stops being a buffer and becomes a floor; `netOut` is grossOut − protocolFee − s; `ironFloor` echoes the attested floor; `topology` and `bridgeUsed` are derived from the hop count and from the fee anchor the Router will use — the same scan of the same producer, so the preview and the executor cannot name different bridges. The integrator rule fits in one sentence and is the most load-bearing advice in this paper: derive your minimum from your own price expectation, never from a preview field.

The Exact Pass. For a pool-exact preview of a concentrated venue, the Quoter calls the pool's real swap entry; the pool calls back demanding payment; the Quoter's callback, instead of paying, reverts — carrying the computed deltas in the revert payload. The revert unwinds all state, nothing is paid, and the decoded output is the venue's own swap over live state, run and rolled back; V4 is handled identically through an unlock whose callback reverts with the packed delta. A pool that refuses the dry run keeps the plan's own claim scaled by the chord below it — never above, because an AMM's output is concave. Hop 0 is capped at what the plan committed, so a capacity-clamped route is not dry-run at a size the Router will never spend. Since 2026-09-03 the returned scalar is net of the protocol fee, deducted once as the preview deducts it, so a minimum derived from it is honoured by the floor (`test/QuoterExactNetOut.t.sol`). This is the Quoter's preview-side instrument; the surface that binds is the in-frame re-derivation.

§12.2 How a quote ages

A quote is a promise about a future block. The study takes it the way an integrator does — `previewAndEncode` returns the preview and the calldata — lets the world move (zero to three trades by someone else through the same pools, each up to 3 % of the shallow reserve, all in the user's direction so every one of them hurts), lets zero to ten seconds pass, and executes the calldata unchanged. 240 samples on a three-token universe of constant-product mocks (A, B = bridge coin, C; direct into the bridge, two hops through it, two hops the other way), every outcome classified (`test/QuoteDelayStatistics.t.sol`):

DRIFT BETWEEN QUOTE AND EXECUTION	SAMPLES	SETTLED	REFUSED BY THE FLOOR (`ROUTERE(5)`)	DELIVERED / PREDICTED — MEAN	MIN
none	64	64	0	10,000 bps	10,000
1 – 100 bps	43	43	0	9,952 bps	9,803
100 – 300 bps	133	102	31	9,966 bps	9,653

DELAY BETWEEN QUOTE AND EXECUTION	SAMPLES	SETTLED	REFUSED (`ROUTERE(5)`)	DELIVERED / PREDICTED — MEAN	MIN
0 s	17	15	2	9,989 bps	9,857
1 – 5 s	99	89	10	9,959 bps	9,653
6 – 10 s	124	105	19	9,983 bps	9,659

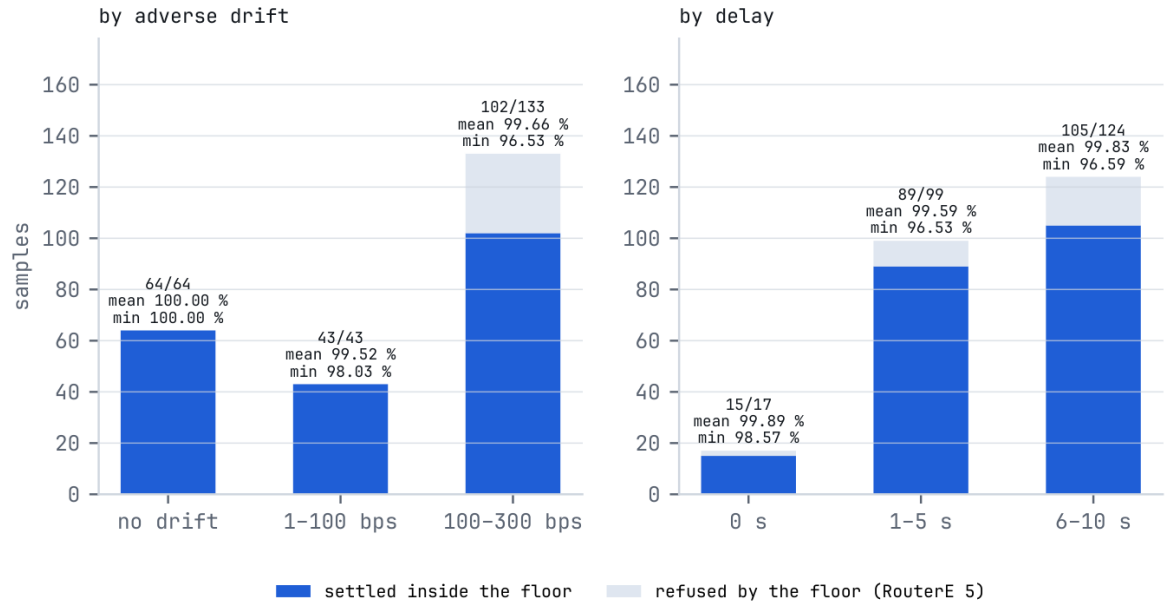


FIG. 8 How a quote ages: settled and refused samples by drift bucket, with the delivered/predicted ratio of every settled sample; the same 240 samples bucketed by delay show no trend.

Time does not move a quote; drift does. Asserted on every sample: a drift-free quote settles at every delay and delivers exactly its prediction (64 of 64); a settlement never delivers below the floor the preview attested; there is no third outcome. After the deadline (`deadline = now + 5`, executed at `+10`) 20 of 20 are refused with the deadline's own code, `RouterE(4)`, and none settles. Under up to 1 % of adverse drift every quote still fills, within 2 % of its prediction; under 1–3 % the floor refuses one in four rather than fill it below the attested output, and the fills land within 3.5 % of the prediction — the sandwich curve of §14 seen from the quote's side.

Live Base (`test/fork/QuoteDelayFork.t.sol`, 1,000 USDC → WETH, real pools, the protocol deployed on a fork at the current block): executed 0, 3, 6 and 10 s later with nothing else moving, the quote delivered exactly its prediction all four times. With 10,000 · 50,000 · 200,000 · 1,000,000 · 5,000,000 USDC traded ahead of it through the same route and the stale calldata executed 10 s later, all five settled inside the floor at 9,999 · 9,998 · 9,994 · 9,974 · 10,000 bps of the prediction: a million dollars ahead of a thousand costs the thousand a quarter of a percent, and the five-million trade routed through pools the thousand-dollar route does not touch. The distributions above are over mocks; live Base gives reach on one pair at one block, not a distribution.

§12.3 What a quote costs through the ABI

Gas measured around the external call, forty sizes each on the three-token universe (`test/QuoterGasStatistics.t.sol`) and five sizes on live Base (`test/fork/QuoterGasFork.t.sol`):

WORLD	`PREVIEWPLAN` MEAN	MIN	MAX	Σ	`PREVIEWANDENCODE` MEAN	Σ
mocks — discovery: the pair known to the factory only, one pool found	121,313	116,468	123,491	1,433	123,317	1,851
mocks — fresh registry: three seeded pools priced	235,574	227,205	237,843	1,815	239,647	1,937
live Base — cold: registry empty, admitted factories swept	1,524,221				1,525,341	
live Base — warm: fresh after one execution, two pools registered	1,414,633				1,416,037	
live Base — cold again after the discovery TTL	1,359,925				1,361,582	

`batchQuote` of ten entries on mocks: 2,055,390 gas, 205,539 per entry. The fresh-registry mock costs more than discovery because it prices three pools where discovery found one; the honest comparison is the live one, where the discovery sweep is about 7 % of a quote — and a quote is a view call an integrator never pays for on chain.

§13 Fee policy: the two regimes and the Surplus Rule

Plain: the protocol takes 0.28 % once, from one measured amount, in a coin the treasury wants to hold, and counts that it did. *Precise:* `PROTOCOL_FEE_BPS = 28` is a compile-time constant in the Core with no setter anywhere in the deployed bytecode; the fee is `mulDivUp(base, 28, 10,000)` — rounded up, so a zero fee is unreachable for any non-zero base — split `TREASURY1_SHARE = 3,000` bps to the first treasury and the remainder to the second (computed as `fee - t1`, so the two always sum to the fee with no dust), the treasuries fixed at construction and freezable for ever by the One-Way Door. No key can raise the rate against a trader and no key can lower it for a favoured integrator: the fee question was answered at compile time and removed from the list of things a key can do.

§13.1 Where the fee is charged: two regimes, both named

The rule since 2026-08-22 is one sentence: **charge once, on the first bridge coin the Router holds.** The Router scans the route for the first hop whose input token is a registered bridge (`hub.isBridgeToken`, in hop order) — that hop is `feeHop` — and charges 28 bps of that hop's *measured* input: on hop 0 the measured pull capped by what the route committed, on a later hop the Router's balance of the bridge coin above its foreign baseline, the quantity already resistant to fee-on-transfer and to stray balances. The Router then charges nothing else on that route. A direct route whose destination is itself a bridge coin (TOKEN → WETH) charges on the output instead, after the floor check — the floor validates swap quality, the protocol's cut comes out of the already-validated amount, and the user's minimum is compared against what they actually receive. This is the **anchored regime**. Why on the bridge and not on the destination: the bridges are WETH and USDC, and the treasury receives a liquid coin it wants to hold instead of dust of whatever tail token is the destination. Why on hop 1's input and not hop 0's output: they are the same token and the same amount, but hop 1's input is measured by the real balance, which is the measurement the Router already makes.

The rule has a second regime the sentence did not name until 2026-09-05. A hand-built route through pools the registry would not hold, with no bridge coin as any hop's input, never finds a `feeHop`; the predicate that charges is then true for *every* hop, and each hop pays 28 bps on its own measured input. This is the **exhaustion regime**, and it is a deliberate rule, not an oversight: charging such a route once, on hop 0, was tried inside the suite on that date and reopened the escape the exhaustion policy exists to close — a value-less first hop carries the fee spot onto dust and the real hop pays nothing (`test/FeeEscapeViaJunkPrefix.t.sol`); five pinned tests refused the change within the run. There is no index at which to insert dust that escapes every hop, which is what "immunity by exhaustion" means. The Solver builds every multi-hop route through registered bridges, so a Solver-built route always pays exactly once; the exhaustion regime is what a caller meets when they route around the registry, and the Quoter models it (`_pack` charges `hops.length` times when no hop input is a bridge, once otherwise), so the preview and the delivery agree in both regimes (`test/ExhaustionRegimePreviewParity.t.sol`).

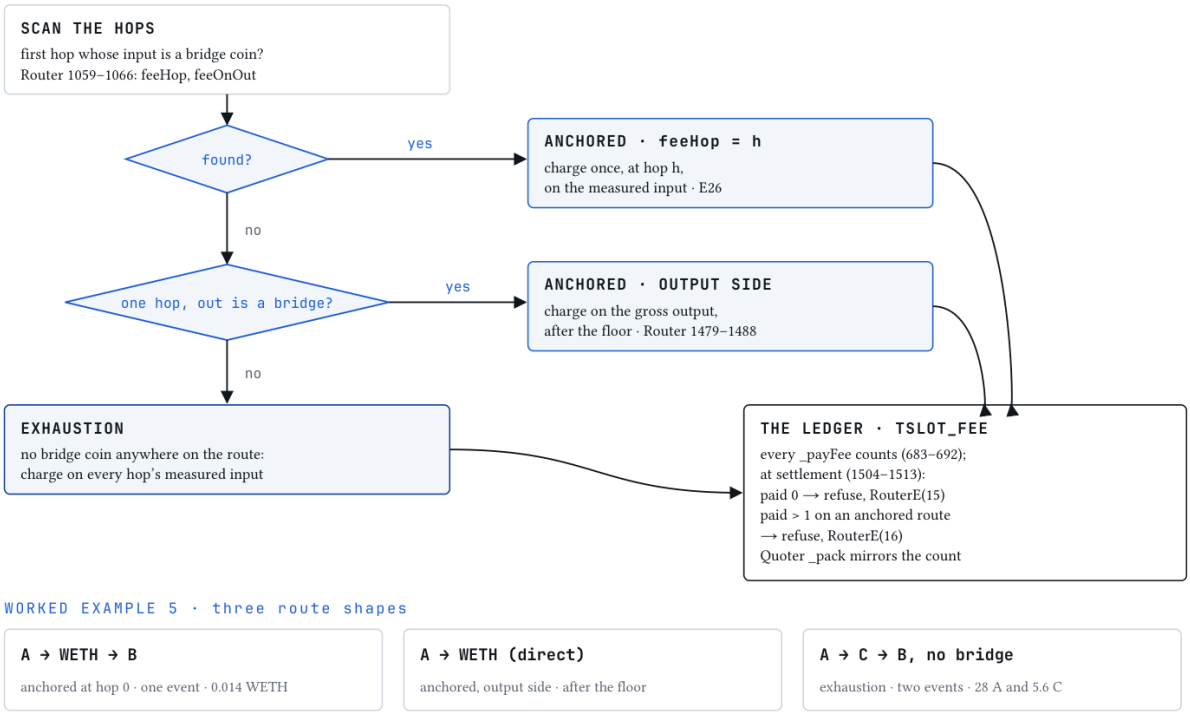


FIG. 9 The two fee regimes as the Router decides them: scan the hops for the first bridge-coin input; anchored routes pay once at that hop (or on the output of a direct route into a bridge coin); routes with no bridged input pay once per hop; the ledger counts and refuses zero, and refuses two on an anchored route.

§13.2 The commitment has one producer, and the contract counts

Two seals accompany the regimes. The sum of a hop's declared leg inputs — the fee base at hop 0 and the scale's denominator — is computed by one function, `_legSum`; before 2026-09-05 each summed the legs in its own loop, two producers of one number. And the fee ledger: `_payFee`, the one function that transfers the fee, counts into a transient slot (`TSLOT_FEE`); every settlement reads the slot once, refuses a delivery that paid nothing (`RouterE(15)`) and, on an anchored route, a second payment (`RouterE(16)`), then clears it. The ledger's two checks are *belts*: in isolation no test can make them fire, because the predicate in front of them leaves no path that settles without paying or pays twice on an anchored route. They are the contract refusing at run time what the tests refuse at review time, on paths that do not exist yet; they are listed as belts, counted as covered nowhere, and absent from the mutation guard, which admits only mutants a named test kills.

§13.3 The Surplus Rule

The **Surplus Rule** is the alignment argument of the fee design, stated as the code implements it: the protocol's take is 28 bps of one measured base, once on an anchored route and once per hop on an exhausted one, and there is no term anywhere in settlement that scales with the difference between what was quoted and what was delivered. Where the base is an input — every multi-hop route and every direct route whose destination is not a bridge coin — the output is untouched: everything the pools deliver above the quote reaches the recipient in full, and the fee cannot rise because a fill came in favourable. Where the base is the output — a direct route into a bridge coin — the fee is 28 bps of the gross delivery, surplus included, and noth-

ing more. The axis on which an aggregator is most tempted to skim is the gap between quote and fill; this design gives that gap no fee term at all, so the headline rate is the true and only rate. The base is never read from calldata: `route.totalOut` reaches the Router as a stranger's claim and is read by nothing (`test_FeeBase_IgnoresLiedAboutTotalOut`).

Worked example 5 — the two regimes on one route. Bridges are {WETH, USDC}; the trader sells 10,000 A.

Anchored, two hops. A → WETH → B. Hop 0 is not charged (its input A is not a bridge). Hop 0's pools pay out 5.000 WETH, measured as the Router's WETH balance above its foreign baseline. `feeHop` = 1: the fee is $\lceil 5.000 \times 28/10{,}000 \rceil = 0.014$ WETH, split 0.0042 to the first treasury and 0.0098 to the second, and hop 1 spends 4.986 WETH into B. One `Fee` event; the ledger reads 1 and is satisfied; the preview deducted 28 bps once.

Anchored, direct into a bridge. A → WETH in one hop. Nothing is charged on the input. The pools deliver 5.000 WETH gross; the floor is checked on 5.000; then the output-side fee $\lceil 5.000 \times 0.0028 \rceil = 0.014$ WETH is paid and 4.986 WETH is delivered; `userMinOut` is compared with 4.986. One `Fee` event, denominated in the output token.

Exhaustion, two hops. A → C → B where none of A, C or B is a bridge — a route no Solver builds. Hop 0 pays $\lceil 10{,}000 \times 0.0028 \rceil = 28$ A and sends 9,972 A into its pools, which pay out 2,000 C; hop 1 pays $\lceil 2{,}000 \times 0.0028 \rceil = 5.6$ C and sends 1,994.4 C into B. Two `Fee` events; the ledger reads 2 with no `feeHop`, so `RouterE(16)` does not apply; the effective rate is $1 - 0.9972^2 = 55.9$ bps, and the preview said so.

§13.4 The fee, measured from outside the Router

`test/FeeSeals.t.sol` reads none of the Router's numbers: the fee token comes from the rule and the bridge list, the base from the pools' balance deltas (what left the Router into the fee hop, or what the previous hop's pools paid out) or the recipient's, the fee from the treasuries' deltas, the count from the `Fee` events — over every shape the Router accepts (one to three hops; bridge in no, first, middle or last position; one or two legs per hop), fuzzed amounts from 10^{12} to 10^{21} wei, 2,000 runs, no failure, the Router holding nothing after every settlement.

ROUTE SHAPE	REGIME	WHERE THE FEE LANDS	MEASURED
A → B (neither a bridge)	exhaustion, one hop	hop 0's input, once	fee = ⌈28 bps × input⌉, one event
W → A (bridge in)	anchored	hop 0's input	same
A → W (bridge out, direct)	anchored, output side	the output	fee = ⌈28 bps × gross output⌉, delivered = gross − fee
A → C → B (no bridge)	exhaustion, two hops	each hop's measured input	two events
W → A → B	anchored at hop 0	hop 0's input	one event
A → W → B	anchored at hop 1	the bridge coin hop 0 produced	one event, ⌈28 bps × what hop 0's pools paid out⌉

ROUTE SHAPE	REGIME	WHERE THE FEE LANDS	MEASURED
A → C → W (bridge only as output of two hops)	exhaustion	each hop's input	two events
A → C → D → B · A → W → C → B · A → C → W → B	as above	as above	3 · 1 · 1 events

How often the fee tests notice a defect was measured rather than assumed (`docs/assurance/fee-seal-detection.json`): every fee mutant — the regime predicate moved either way, the commitment producer, the ledger's two checks and the three fee mutants of the invariant study — was run under twenty fuzz seeds per fuzzed test and once per deterministic test; the detection rate is reported with a Wilson 95 % interval and, where nothing was missed, the rule-of-three bound on the miss probability.

MUTANT	FEESEALS FUZZ	ROUTER CAMPAIGN (2 HOPS, 2 LEGS)	COVERING ARRAY T = 2	JUNK-PREFIX ESCAPE	EXHAUSTION PREVIEW PARITY
exhaustion charges hop 0 only	20/20 [0.84, 1.00]	20/20 [0.84, 1.00]	no	yes	yes
exhaustion skips hop 0	20/20 [0.84, 1.00]	20/20 [0.84, 1.00]	no	yes	yes
commitment counts the first leg only	20/20 [0.84, 1.00]	20/20 [0.84, 1.00]	no	no	no
fee doubled	20/20 [0.84, 1.00]	20/20 [0.84, 1.00]	yes	yes	yes
input-side fee never charged	20/20 [0.84, 1.00]	20/20 [0.84, 1.00]	no	yes	yes
fee charged on both sides	20/20 [0.84, 1.00]	20/20 [0.84, 1.00]	no	yes	yes
belt: settlement without a fee no longer refused	0/20 [0.00, 0.16]	0/20 [0.00, 0.16]	no	no	no
belt: anchored double payment no longer refused	0/20 [0.00, 0.16]	0/20 [0.00, 0.16]	no	no	no

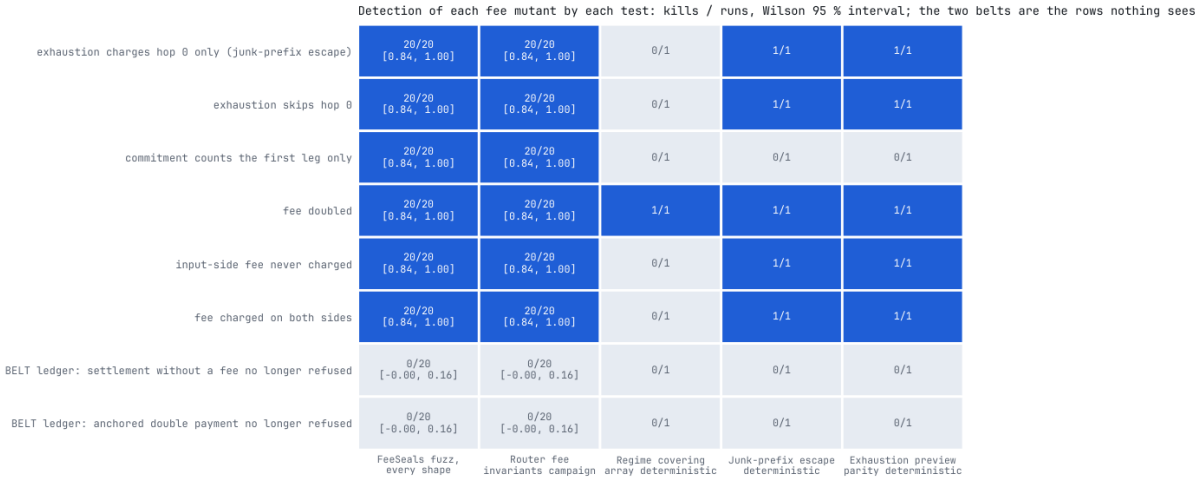


FIG. 10 Detection rate of each fee mutant by each test, as a heat map: kills over runs with the Wilson interval; the two belts are visible as the rows nothing sees.

Read across a row: which tests see this defect. Read down a column: what a test can and cannot see. The first measurement found the Router campaign blind to the exhaustion mutant that spares hop 0 and to the commitment producer, because it built direct one-leg routes only; the campaign was widened the same day to walk two hops and split hop 0 across two pools, with a bridge coin in the universe so one campaign holds every regime, and both are now seen 20 of 20. The covering array sees only the doubled fee, because its rows always cross a bridge. Twenty seeds at 20 of 20 bound a single campaign's miss probability at 15 %, which is why the mutation guard runs the named test and the campaign both.

Summing the section in the form this paper uses — claim, enforcement site, limit: nobody can move the rate or the split, because neither has a setter in the deployed bytecode; a caller cannot shrink the fee by understating a quote, because the base is a measured balance and `route.totalOut` is read by nothing; a caller cannot dodge the fee with a value-less prefix, because a route with no bridged input pays on every hop; and a settlement cannot pay nothing or pay twice on an anchored route, because the Router counts — with the limit that the counter's own two checks are belts no test can make fire.

§14 Security model

Plain: the Router holds nothing between transactions, every swap is all-or-nothing, and each attack class this repository has considered is answered by a named check or named as out of scope. *Precise:* the threat catalogue is `docs/assurance/threats.json`, 23 classes drawn from the public record of exploits; each names the guard symbol that refuses it, and a CI script fails the build if the symbol leaves `src/` or the named test leaves `test/`. Coverage of a catalogue is a floor on what has been considered, never a ceiling on what exists.

§14.1 Attacks and the checks that refuse them

CLASS (ID)	THE CHECK THAT REFUSES IT	WHERE IT IS ENFORCED	STATUS	THE TEST THAT SHOWS THE CHECK CAN FAIL
Reentrancy (DASP-1)	transient lock `nrEntrant` on every value-moving door; the callback surface authenticates its caller	Router `nrEntrant` → `RouterE(7)`; `fallback` → `RouterE(6)`	blocked	`invariant_reentrancyBlocked` (`test/RouterAdversarialV4FromV1.t.sol`)
Access control (DASP-2)	four roles; every privileged door gated; the control tier renounceable for ever	Hub `onlyAdmin` / `onlyControl` → `HubE(1)`; Router `onlyControl` → `RouterE(1)`	blocked	`test/HubAllowHookAfterRenounce.t.sol`, `test/ControlEvents.t.sol`
Arithmetic over/underflow (DASP-3)	checked arithmetic throughout; every `unchecked` block carries a written bound; the hard curves' representability tested at the boundary (`MAX_SQRT_PRICE_MINUS_ONE`)	Core	blocked	`test/BoundaryEquality.t.sol`, `test/CompilerPanicSurface.t.sol`
Denial of service by unbounded iteration (DASP-5)	every route dimension bounded: `MAX_HOPS = 3`, `MAX_LEGS_PER_HOP = 5`, `MAX_LEGS = 11`, `MAX_CANDIDATES = 8`, `MAX_SLOTS = 16`, `MAX_FACTORIES = 16`	Router `_execute` → `RouterE(3)`; Solver; Hub → `HubE(4)` (factories), `_canInsert` (slots)	blocked	`test/RouteHopCeiling.t.sol`
Transaction-order dependence / sandwiching (DASP-7)	the Iron-Law floor, hard-clamped at `FLOOR_HARD_MAX_LOSS_BPS = 2,000`, enforced independently of the caller's minimum, which may only tighten it	Core `ironFloorBpsShv`; Router `_execute` → `RouterE(5)`	blocked (bounded, §14.3)	`test/regime/SandwichCurve.t.sol`; `invariant_DeliveredNeverBelowTheProtocolFloor`
Cross-function reentrancy through a venue (SWC-107)	the same transient lock spans the whole swap, pool callbacks included	Router `nrEntrant`	blocked	`test/V4LockedRegionReentrancy.t.sol`
Price-oracle manipulation (SOK-ORACLE)	there is no feed: the reference quote is produced in the executing frame from the venues traded, and the floor is derived from it (`ironFloorBps`)	Core; Router `_execute`	blocked	`test_ParitySplitRoute_AttestedFloorEqualsEnforcedFloor` (`test/FloorParitySolverRouter.t.sol`)

CLASS (ID)	THE CHECK THAT REFUSES IT	WHERE IT IS ENFORCED	STATUS	THE TEST THAT SHOWS THE CHECK CAN FAIL
Caller-supplied quantity substituted for a measured one (SOK-FEE-BASE)	the fee base is a measured balance (`_chargeHopFee`); `route.totalOut` is read by nothing	Router `_chargeHopFee`	blocked	`test_FeeBase_IgnoresLiedAboutTotalOut` (`test/BlazePhoenixRouter.t.sol`)
Fee-on-transfer and rebasing tokens (SOK-FOT)	every amount that matters is a measured balance delta (`balanceOf`), never an assumed transfer amount	Router, every seam	blocked	`test/FotFloorReprice.t.sol` , `test/PartialFotAtPrePulledDoors.t.sol`
Hostile venue admitted to the registry (SOK-VENUE-ADMISSION)	admission is curator-only, one row per address; the derivation origin is attested at admission (`factoryDeployer`) and frozen after renunciation	Hub `addFactory` , `_probe`	blocked	`test_C4_FirstPin_AfterRenounce_MustNotAttestTheLiveAnswer` (`test/T19ReadmissionEdge.t.sol`)
Admitted venue whose logic moves behind a proxy (SOK-PROXY-VENUE)	re-admission after renunciation refused when the runtime moved (`factoryCodehash`); where a proxy can move its answer without moving its code, the answer is pinned instead	Hub `addFactory` → `HubE(1)`	blocked	`test/FactoryCodehashPin.t.sol` , `test/RenouncedFactoryRearm.t.sol`
Venue row converted from derived to asked after ossification (SOK-MODE-FLIP)	a derive row cannot become an ask row once `controlRenounced` ; tightening in the other direction stays allowed	Hub `addFactory`	blocked	`test_C4_S3_DeriveRowMustNotBecomeAnAskRowAfterRenounce` (`test/T19ReadmissionEdge.t.sol`)

CLASS (ID)	THE CHECK THAT REFUSES IT	WHERE IT IS ENFORCED	STATUS	THE TEST THAT SHOWS THE CHECK CAN FAIL
Published metric reporting the caller's declaration (SOK-VOLUME)	the registry is handed the amount the hop was able to spend, from the measured-over-declared ratio (`hopScale`)	Router `_recordHits`	blocked	`test_INV_F4_VolumeInEqualsTheMeasuredPoolDelta` (`test/VolumeEventFidelity.t.sol`)
Preview that does not predict execution (SOK-QUOTE-EXEC)	preview and delivery asserted equal in both fee regimes (`previewRoute` , `_pack`)	Quoter; Router	blocked	`test_INV_F2_PreviewPredictsDeliveryWithNoBridge` (`test/ExhaustionRegimePreviewParity.t.sol`)
Deterministic derivation steered to an attacker's address (SOK-V4-DERIVE)	the CREATE2 preimage has one producer (`create2Address`); a pool id derives from its own key; `hasCode` and the liveness proof discard what does not resolve	Core; Hub `_probe` , `_scanV4`	blocked	`test/ConditionAdequacyCore.t.sol` , `test/KindIsDerivedNotDeclared.t.sol`
Malicious hook on a hooked venue (SOK-HOOK)	three layers, §14.2: immutable address bits, allow-list with codehash pin (`hookAllowed` , `isHookLive`), the projector Ξ	Router → `RouterE(9)` ; Hub `allowHook` → `HubE(8)` ; Solver `_topKPools`	blocked	`test/V4SievedHookIsTheExecutedHook.t.sol` , `test/SeedPoolHookMustBeAllowed.t.sol` , `test/RenouncedHookRearm.t.sol`
Registry exhaustion by repeated admission (SOK-GRIEF-SEATS)	one row per address; a repeated add refreshes in place instead of consuming a seat (`MAX_FACTORIES`)	Hub `addFactory` → `HubE(4)`	blocked	`test/RenouncedFactoryRearm.t.sol`
Arbitrary external call or transfer-from sink (SOK-ARBITRARY-CALL)	the executor never takes a caller-supplied call target; venue interaction is dispatched from a closed set of shapes selected by kind (`_execScaled`), and an unknown kind reverts	Router `_execScaled` → `RouterE(8)`	blocked	`test/regime/HostileVenueMatrix.t.sol`

CLASS (ID)	THE CHECK THAT REFUSES IT	WHERE IT IS ENFORCED	STATUS	THE TEST THAT SHOWS THE CHECK CAN FAIL
Signature replay on delegated approval (SOK-SIG-REPLAY)	delegated approval is handled by the canonical Permit2 (`permitTransferFrom`); the system holds no signature scheme of its own	Router `swapExact-InWithPer-mit2`	blocked	`test/RouterPermit2OneStep.t.sol`
Gov-ernance capture (SOK-GOV)	there is no governance: no token vote, no timelock, no proxy; the only privileged powers are venue admis-sion and pausing, both renounceable	—	out of scope	—
Malicious upgrade (SOK-UP-GRADE)	the contracts are immutable; there is no proxy and no implementation slot	—	out of scope	—
Com-promise of an extern-al bridge or venue (SOK-BRIDGE-EXT)	third-party venues are composed, not controlled; their failure is bounded here by the floor and the caller's minimum, and their internal security is not claimed	—	out of scope	—
Com-promise of a user's private key (SOK-KEY)	outside the trust boundary; recorded because the published incident data attribute a large share of losses to it	—	out of scope	—

Nineteen of the twenty-three classes are answered by a named guard; four are out of scope with the reason recorded. The catalogue is cross-checked from the other direction by an asset closure (`assets.json`): six things the system holds or decides — funds in transit, delegated allowance, the protocol fee, registry integrity, stranded balances, caller gas — crossed with five ways a thing can be lost, thirty cells, each resolving to a threat class, an out-of-scope rationale or an explicit *open*. Nineteen cells are covered, three are out of scope, five are not applicable, and three are open and published as such: griefing the fee's denomination, stranding dust to make recovery uneconomic, and a caller meeting one of three reachable compiler panics instead of a named refusal. None of the open cells is a path to anyone else's funds.

§14.2 Hostile hooks: three layers, cheapest first

A hook is arbitrary code running inside the settlement the Router is performing, and the naïve defence — asking it what it will do — is no defence, because a hostile hook lies or re-enters. Three independent layers stand instead, arranged so the cheapest and most certain runs first; they bound delta alteration, and the one residual they do not reach — a delta-free hook overriding its dynamic fee (§5.4) — is bounded by Φ and the caller's minimum instead.

Layer 1 — the permission bits live 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 fixed at deployment by the CREATE2 salt, and the PoolManager itself checks the same bits before invoking the hook, so a hook cannot have a permission its address does not declare. Exactly two bits allow a hook to modify swap accounting, and the Router masks them with a single AND before any token moves (`hookAltersDeltas`):

$$(\text{uint160}(\text{hook})\ \&\ \texttt{0x3FFF})\ \&\ ((1 \ll 3) \mid (1 \ll 2)) \neq 0 \implies \text{leg refused, } \texttt{RouterE(9)} \tag{E18}$$

No call into untrusted code, nothing to deceive, and evasion would require an address whose bits contradict the manager's own enforcement. A proxy can replace every line of its logic and still gain no bit its address never had — which is why this refusal sits at projection distance zero (§15.9) while the codehash pin beneath it does not.

Layer 2 — an allow-list, default closed, with the code pinned. A hook with clean bits can still revert maliciously to grief or burn gas, so a V4 pool is routable only if its hook is allow-listed by the curator, and the Hub pins the hook's codehash at admission: a hook whose runtime later changes is auto-paused without eviction (`isHookLive` false) and resumes only if re-admitted — a power that, after renunciation, is refused for any hook whose code moved. The pin's limit is stated at the site: it binds runtime code, so it catches a redeploy or a direct mutation and does not catch a delegate proxy whose implementation is swapped while its own runtime stays byte-identical, because the EVM gives a contract no way to read another's storage. Admitting an upgradeable hook is a human judgement this pin cannot replace. The layers that do not depend on it stand regardless.

Layer 3 — the route is never even shown. The projector Ξ removes any pool whose hook alters deltas from the Solver's candidate set before ranking (`_topKPools`), and the Hub's read channel serves only hooks that are live; a route touching an inadmissible hook does not participate in the maximisation. A route-shape rule completes it: canonical order, route-wide — no hookless leg ever executes after a hook has run (§11.4).

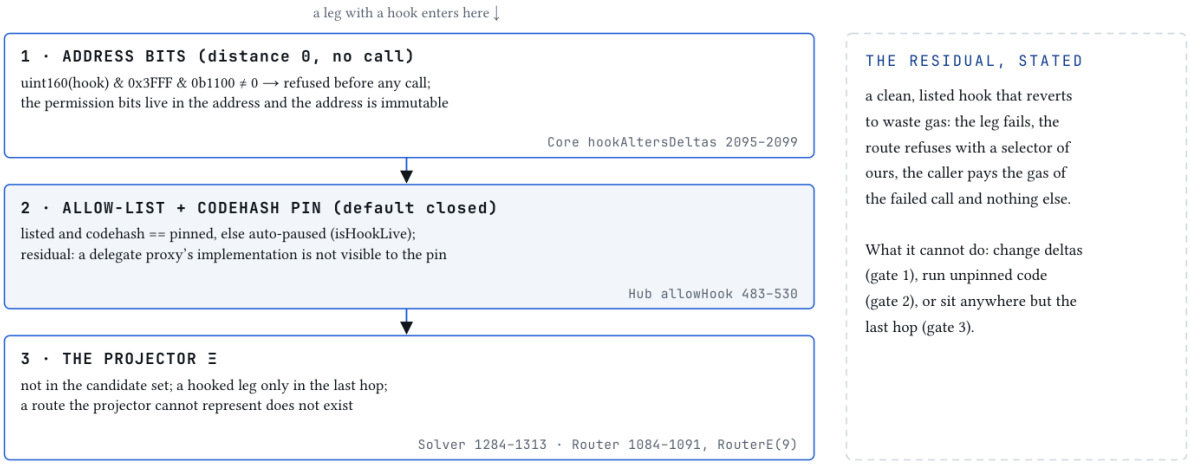


FIG. 11 Three layers against hostile hooks, cheapest first: the address bit-mask (no call into untrusted code), the allow-list with the codehash pin (default closed), the projector Ξ (route not representable). Each layer answers a different capability; the residual — a clean, listed hook that reverts to waste gas — is named.

Defeating the stack requires a hook that carries no delta bits, sits on the allow-list unchanged, and still subverts settlement — a combination Layer 1's appeal to V4's own immutable enforcement rules out for delta alteration; the fee-override residual of §5.4 remains, and the floor bounds it. The residual is stated in the table: an allow-listed, delta-free hook can still revert mid-swap and waste the gas of trying. And the sieve's single lever is named: the sieved hook binds the executed swap because the canonical PoolManager echoes unlock data verbatim and calls back only its unlock caller; the Hub's `setV4Manager`, a control power, is the one address that assumption rests on, and a dishonest manager collapses the measured output so the caller's own bound refuses the swap (`HostileV4Manager` campaign) — bounded, not absent.

§14.3 Bounded, not eliminated: staleness, ordering, and the sandwich curve

A quote is a statement about the past, and the trade happens in the future. The in-frame solving door closes the operator-side half of that gap completely — the route is derived inside the transaction that executes it, from reserves as they stand in that block. What remains is the chain's own half: the pool can move between the moment you sign and the moment your transaction is included. Nobody can prevent this from inside a contract, because ordering is decided before the code runs; a protocol claiming to have eliminated front-running has either moved your trade off the public queue, trusting whoever now holds it, or is describing something narrower than it sounds. What this protocol does is bound it, twice and independently: your minimum caps the damage at the number you chose, and the Iron-Law floor — re-derived by the Router from measured impact and the final hop's in-frame quote, not from anything the attacker or the route supplied — caps it again on the protocol's own account.

The bound is measured from the attacker's side (`test/regime/SandwichCurve.t.sol`): the victim's route and floor are fixed at quote time, as in a pending transaction; the attacker trades a fraction of the pool's depth ahead of the victim, the victim executes, the attacker trades back. The venue is a constant-product pair — the shape every sandwich model uses — so the curve is a property of the floor. On a 1 %-of-depth trade (10,000 against 1,000,000 a side):

ATTACKER MOVES	VICTIM	VICTIM'S LOSS VS THE QUOTE	ATTACKER'S ROUND TRIP
0.1 % of depth	settles	0.47 %	+13.9
0.5 %	settles	1.26 %	+68.9
1 %	settles	2.22 %	+136.7
2 %	settles	4.12 %	+268.7
3 % and beyond	**refused**	0	−174.5 … −544.9

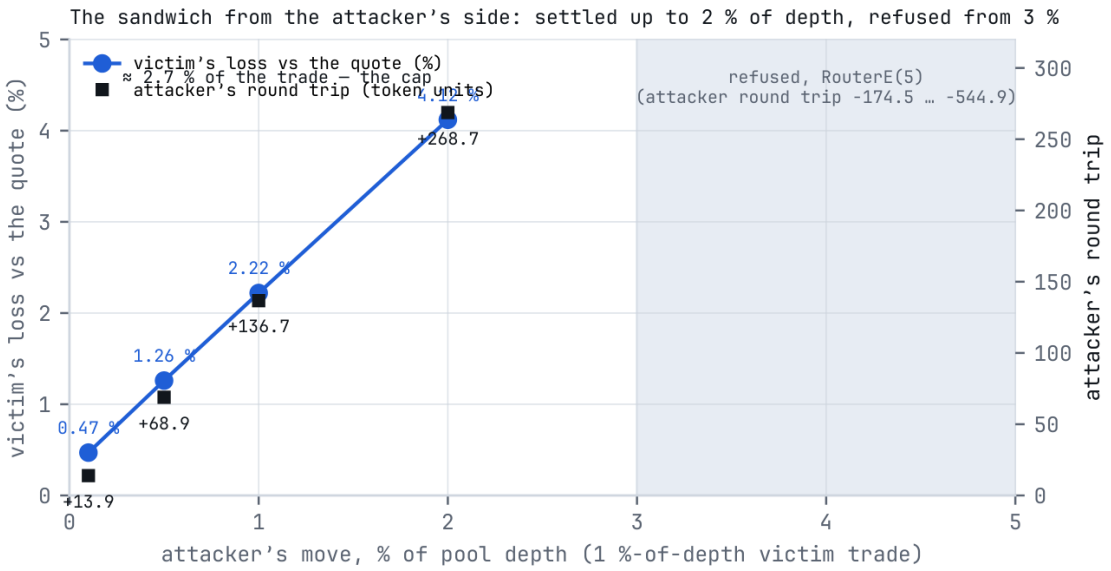


FIG. 12 The sandwich curve from the attacker's side: the victim's loss against the quote and the attacker's round trip, by fraction of depth moved ahead of the victim; the refusal edge near 3 %, closed upward.

The guarantee asserted at every point: a settled victim never receives less than the floor attested at quote time, so the loss is bounded by the distance between the attested quote and the attested floor; and the refusal region is closed upward — past the edge every larger manipulation is refused and the attacker is left holding the price they moved. The number worth quoting is the last settled row: the floor caps what a sandwich can take from a 1 % trade at about 2.7 % of it, and turns the attacker's trade into a loss the moment it would take more. The in-frame route adds a quieter deterrent: there is no pre-published path for a searcher to study, because the path does not exist until the block that executes it. A wide minimum is an invitation; a tight one is the single most effective control in this section that belongs to you.

§14.4 Hostile venues

Venue pathologies have their own matrix (§15.5): ten misbehaviours — a pool that pays nothing, pays half, returns a 64 KiB returndata bomb, burns all gas on read or on swap, a token whose `decimals()` never returns, a payment callback fired twice, a pool that re-enters the Router before paying, a reverting `slot0()`, a factory answering with a pool on other tokens — crossed with the calldata door and the solving door, 20 of 20 cells settling with the delivered amount equal to the balance delta and nothing left on the Router, or refusing with a selector of ours. A swap that burns all forwarded gas is the one cell no caller can decide for its callee; the transaction reverts whole and the user's balance is untouched. Every guarded read the Core makes to a venue is capped at `GAS_CAP = 100,000` gas, so a venue that burns gas on a *read* costs a bounded amount and is dropped, not fatal.

§14.5 The hostile front end

On every aggregator ever built, the interface proposes the minimum, and a careless one proposes it badly — no contract can know what you would have chosen. What a contract can do is make the proposal provable: every swap emits `ExecutionProof` — the quote used, the amount delivered, the floor applied — permanently, on-chain, so any interface's execution quality is a public, per-transaction record that anyone can audit, in-

cluding you. An interface that consistently proposes bad minimums writes its own indictment into the event log. The defence that completes the mechanism is a habit: look at the minimum before you sign; it is the one field that is entirely yours.

§14.6 What a hostile key can and cannot do

Before renunciation, administrative keys exist, and the honest way to describe them is by capability, hostile holder assumed; the full enumeration of powers is §16.4.

What a hostile Router key can do: redirect the fee destination (`setTreasuries`); pause swaps (`setPaused`); repoint the WETH and Permit2 contracts the native and Permit2 doors call (`setWeth`, `setPermit2`), which makes those two doors only as trustworthy as the addresses set; queue and, 48 hours later, execute a rescue of tokens stranded in the Router (`queueRescue`, `executeRescue`, `RESCUE_DELAY = 48 hours`, announced on-chain by `RescueQueued`). **What it cannot do:** move the fee rate, the split or the floors, because they are compile-time constants with no setters; take principal, because there is no proxy, no upgrade path, no `selfdestruct`, no standing approval of user funds to the Router, and the Router holds nothing at rest — the functions that would do it are not disabled, they are absent.

What a hostile Hub key can do: list a venue or a hook (`addFactory`, `allowHook`), seed pools (`seedPool`, through an operator seat), pause the registry's learning (`setPaused`), repoint the Hub's view of the Router, Solver and Quoter (`setRoles`), and repoint the V4 manager (`setV4Manager`, the sieve's single lever). **What it cannot do:** misprice a fill, because the registry decides only what is seen (§9.6) — the sharpest thing a hostile listing reaches is the gap between a quote and *your minimum*; principal is out of reach, your slippage tolerance is not, which is one more reason the minimum is yours to set tightly.

§14.7 The One-Way Door

Administrative authority ends irreversibly, in bytecode, not by promise. Both the Hub and the Router expose `renounceControl`, and each refuses to be called while paused (`RouterE(2)`, `HubE(2)`): a paused-then-renounced protocol would be a terminal state nobody wants and nobody can leave, and pausing is worth doing precisely because control is retained to act. The renunciation flag is written `true` at exactly one site in each contract and written `false` at none, so no function in the deployed bytecode can reopen the door — the difference between "we won't" and "we can't".

What closes and what stays open is enumerated, not summarised. On the Router, everything under `onlyControl` dies: `setAdmin`, `setTreasuries`, `setPermit2`, `setWeth`, `setPaused`, the rescue path — the treasuries, the wiring and the pause flag freeze at their current values for ever, and the Router keeps executing swaps under that fixed configuration. On the Hub, the **control tier** dies: `setRoles`, `setOperator`, `setPaused`, `setV4Manager`, `removeBridge`, and hook *de-listing*. The **curator tier** survives, and only grows the registry: `addFactory` (refusing, after renunciation, any factory whose runtime moved or any derive row that would become an ask row), `addBridge` (add-only, at most three, idempotent), `allowHook(h, true)` (refusing a listed hook whose code moved); existing operator seats keep `seedPool` and `addV4`, and no new seat can be granted. A malicious listing after renunciation cannot drain: pools are validated at quote and at execution and bounded by the floor and the caller's minimum. The registry keeps learning by trading. Anyone repeating "renounced" about this protocol should be able to say which doors stay open — admission, bridges, hook listing, curated seeding — and which close: every power that redirects or freezes.

§14.8 What this model asks of you

The protocol closes what a contract can close: address forgery, callback spoofing, accounting forgery, hostile hooks, arithmetic. It bounds what no contract can eliminate: staleness and ordering, by your minimum and the floor. And it publishes evidence against what it cannot even bound: a front end proposing bad minimums on your behalf. Every row converges on the same field. Set your minimum yourself, deliberately, every time — it is the one protection in this section whose quality depends on nobody's competence but yours.

§15 The verification apparatus, measured

Plain: every number here is printed beside the denominator it was measured against, is recomputed from a clean checkout by a script or a named `forge` command (Appendix C), and is followed by what it does not establish. *Precise:* the tree measured is `main` at `8949a9d` and the branch that became it, on the release profile (`optimizer_runs = 300`, `via_ir`), 2026-09-05.

§15.1 The suite in numbers

WHAT	COUNT	HOW IT IS COUNTED
declared tests	1,609 `test*` / `invariant*` / `check*` functions across 236 `.t.sol` files	`grep` over `test/`
green on the release profile, fork suites excluded	1,319 passed · 0 failed · 1 skipped, of 1,320	`forge test --no-match-path 'test/fork/**'`
green on live liquidity	119 of 119 on the 39 suites of the fork lane, five chains; plus the three live-Base tests of §12	`forge test --match-path 'test/fork/**'` with an archive RPC
stateful invariants	43 `invariant_*` functions in 14 campaigns	`grep -rc 'function invariant_' test`
curated mutants, all killed	268 of 268	`.github/scripts/mutants.py`; `check_targets.py`
regime covering arrays	strength 2: 63 rows, all 258 pairs · strength 3: 168 rows, all 1,636 triples · of 5,184 combinations	`covering_array.py --check`
shipped sizes (runtime bytes; EIP-170 limit 24,576; project gate 24,000)	Hub 23,648 · Router 23,781 · Solver 19,686 · Quoter 11,429 · Core 6,442	`FOUNDRY_PROFILE=release forge build --sizes`

§15.2 The evidence chain, and the curated mutation guard

A property is only as good as the shortest path from a threat to something that would fail if the property broke: `threat → property → guard → test → mutant`. A guard is named by symbol, a test by name, a mutant by the exact line it alters; if any of the three leaves the tree, the build fails. Every regression test in the suite has been seen to fail once, for the reason it is named after — a fix arrives with the test that was red against the code without it.

The mutation guard holds 268 hand-written mutants, each pairing one exact line of source with the single test that must fail once that line is altered — a guard deleted, a comparison flipped at the bound that decides a refusal, an authorisation widened, an error code swapped with its neighbour's. Three properties make it a guard rather than a score: the paired test is run green on the unmutated tree first; a mutation the optimiser removes — identical runtime bytecode — is reported as *inert*, never as killed; and a one-second static check verifies that every mutant still points at exactly one line. 203 of 203 are killed. The figure is adequacy against *this* register: a saturated score is a floor, not a ceiling.

§15.3 Mutants aimed at the invariants

Until 2026-09-05 none of the curated mutants named a stateful invariant as the test that must die to it. The 43 `invariant_*` functions across 14 campaigns were green, and nothing had asked whether any of them could go red: a campaign whose handler never reaches the state a property protects certifies that property over an empty universe, and looks identical from outside to one that reaches it. The measurement is the guard's, pointed at the campaigns: alter one guard in the source, run every invariant on one seed, record which ones notice (`docs/assurance/invariant-mutants.json`).

MUTANT	WHAT IT REMOVES	NOTICED BY
the final transfer pays one wei less	holds-nothing	`RouterHoldsNothing`, `routerHoldsNothing`, `HoldsNothing-BeyondTheSeed`
the protocol fee doubles	the fee ceiling	`FeeNeverExceedsProtocolMax`
the input-side fee is never charged	the fee floor	`FeeNeverEscapes`, `DeliveredNeverBelowUserMinOut`
the fee is charged on both sides	one fee, one side	`FeeIsChargedOnExactlyOneSide`
30 % of every fee goes to a dead address	conservation	`conservationA/B/C/H`, `TokenConservation`, `LedgerConserva-tionA`
the leg-pair guard admits a leg on the wrong pair	homogeneous hops	`DivergentLegNeverSettles`, `HoldsNothingBeyondTheSeed`, `StrandedMoneyIsNeverSwept`
`whenLive` no longer checks `paused`	the pause	`PausedRouterNeverSettles`
the reentrancy lock no longer refuses	the lock	`reentrancyBlocked`
the protocol floor is halved	the floor	`DeliveredNeverBelowTheProtocolFloor` — closed 2026-09-05

MUTANT	WHAT IT REMOVES	NOTICED BY
`MAX_SLOTS` becomes 17	the registry bound	`neverExceedsMaxSlots`
the registration pair-proof is removed	pair authenticity	`ActiveEntriesTradeThePair` — closed 2026-09-05
the input-residual sweep ignores its baseline	stranded money	`StrandedMoneyIsNeverSwept`, `HoldsNothingBeyondTheSeed`
the bridge-residual sweep ignores its baseline	stranded money	the same pair — closed 2026-09-05
the output measurement ignores its baseline	stranded money	the same pair
the post-fee `userMinOut` check is halved	—	survives, by design: a regression sentinel unreachable in campaign universes with no fee-on-transfer output token; two unit mutants watch it

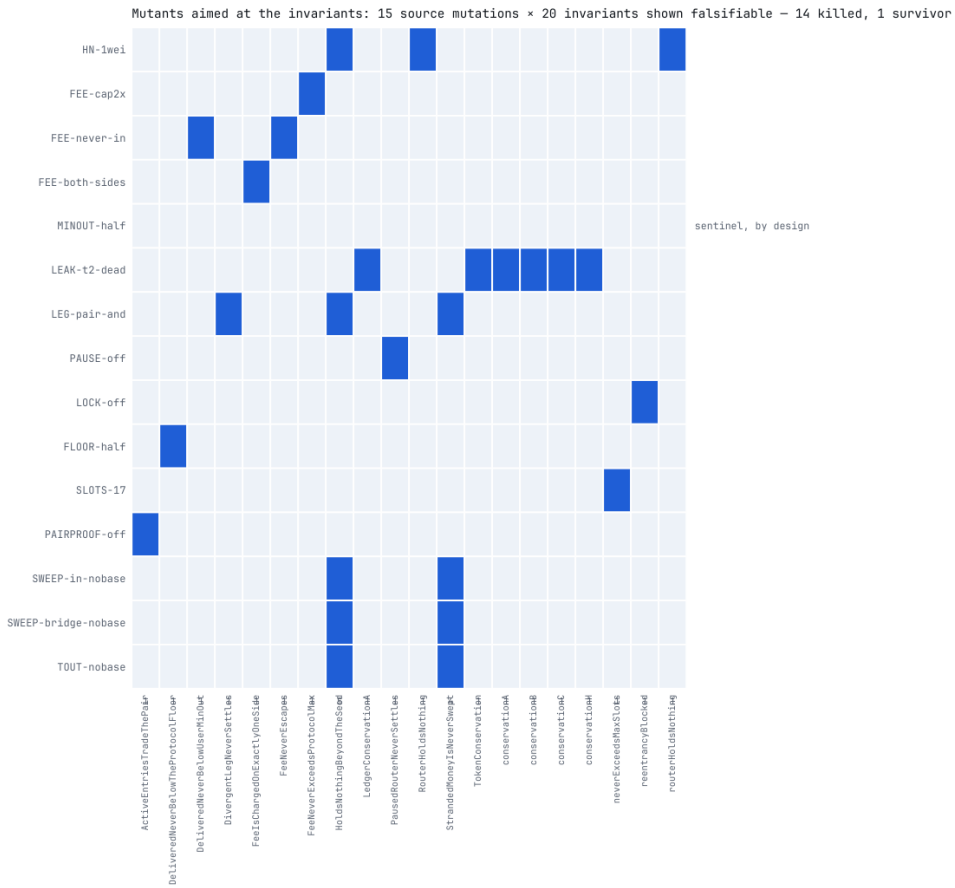


FIG. 13 Mutants aimed at the invariants: fifteen source mutations against forty-three stateful invariants, the invariants that went red per mutant, the three closed on 2026-09-05 by giving a campaign the action it lacked, and the one sentinel that survives by design.

Eleven of fifteen were noticed on the first measurement; twenty distinct invariant names went red at least once. Three of the four survivors were closed the same day by giving a campaign the action it lacked — foreign pools offered under the pair, a two-hop route over a pre-seeded intermediate balance, a whale moving the pool 0–8 % against a quoted route before the stale route executes — each verified red without its guard; two guards with no watcher of any kind was the number this measurement existed to print, and it is now zero. Fourteen of fifteen are noticed; the fourteen are guard entries paired with the invariant that dies to them. A green suite with an unwatched guard in it was the shape of both documented regressions in this codebase; the measurement that finds one now runs on every guard the campaigns claim.

§15.4 Regime covering arrays

Coverage criteria index the code; mutation indexes an injected fault; neither indexes the state a *fixture* fixes before the call. Ten such factors are enumerated — venue family (V2, V3, Solidly), hops (1–3), legs per hop (1–2), whether the input token is a registered bridge, whether the intermediate is, fee-on-transfer shape (none, pull-only, every transfer), the input token's decimals (18, 6), the door (calldata route, solve-in-transaction, Permit2), control (live, renounced) and whether the pair is full — 5,184 combinations. `covering_array.py` generates arrays in which every pair (strength 2) or every triple (strength 3) of factor values appears in at least one row, each row one fixture through one harness (`test/regime/RegimeHarness.sol`) with one assertion: the swap settles, with the delivered amount equal to the recipient's balance delta, at least the floor the Router emitted, and nothing left on the Router — or it is refused with a selector of ours. A panic, a foreign selector, an under-delivery or a stranded balance is a third way and fails the row.

STRENGTH	ROWS	TUPLES HELD	SETTLED	REFUSED, OURS	NOT CONSTRUCTIBLE	THIRD WAY
2	63	258 pairs	53	4	6	0
3	168	1,636 triples	158	10 — `SolverE(5)` ×7, `RouterE(13)` ×3	0	0

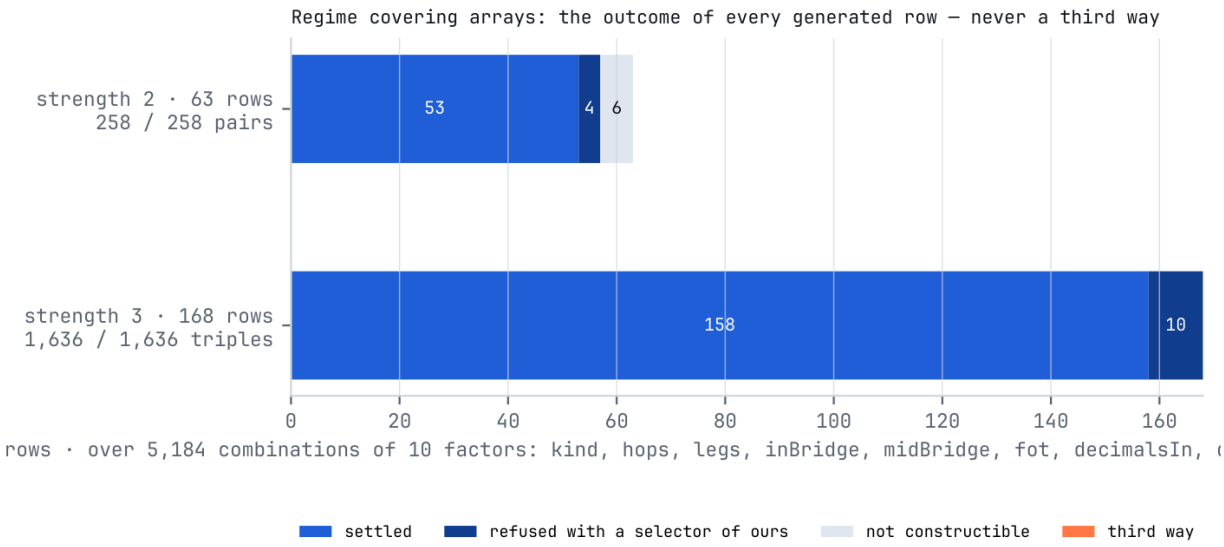


FIG. 14 Regime covering arrays: outcomes of every generated row at strength 2 (63 rows) and strength 3 (168 rows) — settled, refused with a selector of ours, not constructible, third way — over 5,184 combinations of ten factors.

The refusals are the planner having no bridged path to build and the executor declining a taxed token on a concentrated-only route; the six rows at strength 2 that cannot be built (the pull-only-taxed token has no six-decimal form) are printed by name and count against the denominator. The families not in the array — V4, native V4, Algebra, the native door, hooks — are stated in `regimes-covering.json`. The array's first run made a frame explicit that the parity tests had pinned only on one side: the floor the Router enforces equals the attested floor when the fee comes off the output, and sits inside $[\text{attested} \times (1 - \text{fee}), \text{attested}]$ when it comes off the input — both frames are now asserted on every row. The generated file is checked against its generator in CI, so the array cannot drift from the factors it claims to cover.

§15.5 The hostile-venue matrix

Token pathologies had their tests; venue pathologies had none in a matrix. `test/regime/HostileVenues.sol` holds one misbehaviour per venue and `HostileVenueMatrix.t.sol` crosses each with the calldata door and the solving door under the covering array's rule:

VENUE	CALLDATA DOOR	SOLVING DOOR
pays nothing / pays half	refused `RouterE(5)`	refused `RouterE(5)`
returndata bomb on the reserve read	settles	settles
reserve read burns all gas	refused `RouterE(8)`	the planner never selects it
swap burns all gas	whole transaction reverts, balance untouched	whole transaction reverts, balance untouched
`decimals()` burns all gas	settles	settles
payment callback fired twice	refused `RouterE(6)`	refused `RouterE(6)`
re-enters the Router before paying	settles; the nested swap never ran	settles; the nested swap never ran
`slot0()` reverts	settles on the attested quote	the planner never selects it
factory answers with a pool on other tokens	never listed	never listed

Twenty of twenty cells: settle or refuse with a selector of ours, never a third way. The last row is what the matrix's first run changed — discovery listed a pool a curator-admitted factory answered with, and the executor refused it at the seam that pays; an asked pool now proves its own pair before discovery lists it (§7).

§15.6 Canonical oracles

Every mock in the suite quotes with the Core's own formulas, so a defect in a formula is invisible to every parity test that uses them: the oracle is the object. `test/regime/CanonicalOracles.t.sol` holds three implementations written from the venues' published invariants — constant product with the fee on the input, the V3 single-tick square-root-price step with the pool's own against-the-trader rounding, the Solidly stable curve solved by Newton's method — and fuzzes the Core against them, 5,000 runs each, asserting the direction first (the Core never promises more than the venue's mathematics delivers) and the tightness second.

FAMILY	DIRECTION	TIGHTNESS MEASURED
Uniswap V2	the Core never exceeds the specification	exact to the wei
Solidly stable	the Core never exceeds the curve by more than the solver's own last step	within 4 wei
Uniswap V3	the Core never exceeds the specification by more than one unit in the last place (ulp) of the square-root price, worth $L/2^{96}$ wei	below one wei for every pool with $L < 2^{96}$

The V3 bound is stated in the quantity that causes it — the pool rounds its new price against the trader and the Core rounds it once, so the two can differ by one unit of $\sqrt{P}$ — and the assertion is the bound, not the sample (13 wei on a 6.5×10^{34} output at $L = 10^{30}$). The oracles are written from the specifications, not from the venues' bytecode; that is their limit.

§15.7 Metamorphic relations

An oracle written from the same formula cannot catch the formula. A metamorphic relation asks instead how the output must *move* when the input moves, and the venue's own curve answers that without a reference implementation. `test/CoreMetamorphicRelations.t.sol` holds fourteen relations over the Core's quote maths and `test/RouteMetamorphicRelations.t.sol` five over the Solver's plan on a two-pool universe with a real Hub and a real Solver:

LEVEL	RELATION	WHAT IT SAYS
Core, V2 · V3 · stable	MR1 monotone	more in, never less out
Core, V2 · V3 · stable	MR2 sub-additive	splitting an order across the same pool never gains
Core, V2 · stable	MR3 no round trip	in, then back on the updated reserves, never returns more than went in
Core, V2 · V3 · stable	MR4 scale equivariance	scaling order and pool together scales the output, to rounding
Core, V2 · V3	MR5 fee monotone	a higher fee never pays more
Core, V3	MR6 direction symmetry	at price 1.0 the two arms of `outV3` agree to two ulps
Core, Solidly	MR7 identity	the volatile arm *is* `outV2`, one producer
plan	MR-R1	the split never pays less than the best single pool would
plan	MR-R2	monotone in `amountIn`
plan	MR-R3	adding a pool inside the price band never lowers the plan
plan	MR-R4	registration order moves the plan by at most one weight unit of the split
plan	MR-R5	the attested floor never exceeds the expected output

Three of the bounds were measured before they were written, and the number in the assertion is the mechanism, not the sample. MR6: at $L = 8.5 \times 10^{37}$ and 1,374 wei in, one arm quoted 1,073,741,823 wei — exactly $L/2^{96}$, one ulp — while the other quoted zero, its step rounded to nothing, failing closed; the relation states two ulps, one per arm. MR-R1 and MR-R3: with one pool priced 10^{10} away from the other, the plan paid 35 % less than the outlier alone — the depth-weighted median refusing to believe it — so the relations hold over pools the band admits, and the domain keeps the two prices within ±4 %. MR-R4: a pool whose depth weight floors to the minimum is kept as a dust leg when registered first and cut when registered second, moving the plan by 3×10^{-7} of the output — the split's keep-or-cut of a minimum-weight pool depends on its position, and that is now written where it was found.

§15.8 N-version: the same source under other code generation

The suite executes one binary — the one the release settings emit, and `profile_parity.py` keeps the suite on those settings. Every other optimiser setting is a different program compiled from the same source, and the compiler is a component. `test/nversion/` compiles the Core's quote maths under two more profiles — `optimizer_runs` 1 and 20,000, everything else identical — and `NVersionCore.t.sol` deploys the three binaries side by side and asserts equality on fuzzed inputs: `outV2`, `outV3`, `outSolidly`, `outSolidlyStable`, `mulDiv`, `mulDivUp`, 3,000 runs each over the whole input domain. The lane first asserts that the alternate bytecode *differs* from the shipped one, so it cannot pass by comparing a program with itself, and a missing artefact is a failure, not a skip.

The three binaries agree. The compiler was not the finding. The lane's first full-domain run made the *reference* revert: the stable curve's fit check divided `max · WAD / b`, and that division itself overflowed on a pool holding under one unit of a token; a second escape — a whale input reserve against a dust output reserve — passed the invariant check and overflowed the derivative. Neither was a wrong number; both were reverts inside a quote, and `universalQuote` is a library `DELEGATECALL`, so a registered pool holding one wei of the output token unwound the Solver's plan for its pair. The repair (§6) asks the question the division stood in for and went in red-first; a further guard — a cap on Newton's iterate — was written, found unobservable by any test once the two checks hold, and removed rather than carried. Its class is the one this repository names first — fail-open — found by an instrument that was not looking for it.

§15.9 Projection distance

Every defect confirmed in this repository, internal or reported, has one shape: a check observes one object while a *different* object decides the behaviour. That makes the interesting property of a refusal not its complexity but its distance — how far what it reads sits from what decides. `projection_distance.py` classifies the predicate guarding every `revert` in the five contracts: a constant, an immutable, a balance delta or a quantity computed in-frame from one is distance 0; a field of the caller's route, or a lookup in another contract's mutable storage, is distance 1; a property of a contract that can delegate its behaviour is unbounded. Over 99 refusal sites: **86 at distance 0, 11 at distance 1, 2 unbounded.** The two unbounded sites are both `EXTCODEHASH` pins — an admitted hook's code and an admitted factory's — and they are not symmetric: for hooks the residual is closed by the address bits the pin does not carry (§14.2); for factories there is no equivalent, and the residual is bounded by pool admission, the per-leg floor and the user's minimum, so the damage is route degradation rather than drainage, documented at the site. The screen's limit: it reads one predicate

and cannot see that a value was constrained three frames up; a caller-declared field that is also the pool selector is at distance 0 only because one assignment binds them, and that assignment is pinned by `test/V4SievedHookIsTheExecutedHook.t.sol`.

§15.10 Evidence independence, the shared-quantity register, and the artefact

For each load-bearing guard the apparatus records how many *independent* confirmations exist — by source, oracle, tool, environment and methodology — and reports the minimum, because a chain of evidence fails at its weakest link; load-bearing properties are confirmed by at least two routes that share no hypothesis. The shared-quantity register (`SHARED_QUANTITIES.md`) enumerates every quantity with more than one producer or consumer, states the *question* it answers, and grades what binds the copies — `SINGLE`, `PINNED`, `WEAK`, `OPEN`, `UNVERIFIED` — with a CI rule that demotes a row whose test does not reach what it claims to pin; every two-producer quantity in the relational register (7 of 7) is tied by a named test. And the compiled artefact is read, not only the source: opcodes that must never appear (`SELFDESTRUCT`, `CALLCODE`, `ORIGIN`); the transient opcodes still present in the executor, so the reentrancy lock has not quietly become persistent storage; every declared function present in the dispatcher; three of the five contracts emitting zero storage writes; 88.3 % of the shipped-shape instruction stream proven executed by a sound lower bound, verified against a ground-truth contract the instrument must classify correctly before it prints; and, for one recorded honest V2 swap replayed on the release Router with the opcode checked at every step, 4,382 Router instructions and 407 Hub instructions run, zero mismatches — 28.2 % and 2.5 % of the two code sections, the complement being code no recorded scenario ran, not dead code.

§15.11 What none of this establishes

- **No probability of correctness.** The literature on validating ultra-high dependability is explicit that testing cannot produce one; none is stated here.
- **Mutation adequacy is adequacy against this register.** It is hand-curated; a saturated score is a floor.
- **Threat and regime coverage are floors on what has been considered.** Classes and factors nobody has named are outside the denominator by construction.
- **The distributions of §12 are over mocks.** Live Base gives reach, not a distribution.
- **Detection rates are per test at 256 fuzz runs per seed.** A test that misses at that budget may catch at a larger one.
- **This repository is V2, and V2 is not deployed** (§19).

Every instrument above was calibrated before its number was printed: run first against a defect already known by another route, and corrected where it missed it — a text matcher that read documentation as behaviour because it matched text inside comments, a coverage regex that did not allow for a call-option block, a counterpart search whose pattern list was empty. An instrument is worth nothing until it finds the instance already known; a verdict without a search is vacuity; and anything the compiler decides is measured, never estimated.

We know of no other public DeFi codebase that publishes a curated, named mutation guard with per-mutant killing tests and inert-mutation reporting; a shared-quantity register with a CI rule that demotes it; mutants aimed at its stateful invariants with the survivor published; a strength-3 covering array over its regime

factors under one assertion; N-version testing of its quote maths; a measured curve of how its quotes age; or the detection rate of its own fee mutants with a confidence interval. The artefacts are in the repository; if you find a prior instance, the claim is wrong and we would like to know.

§16 Architecture and sizes

Plain: five contracts, one of which is pure mathematics imported by the other four, and no piece of mathematics is defined twice. *Precise:* the Core is a stateless library with no storage, no owner and no upgrade path; Hub, Solver, Router and Quoter import it and none redefines a primitive, so one audit of the mathematics covers every consumer.

§16.1 Five contracts: what each holds, what each may call

CONTRACT	HOLDS	MAY CALL	WRITES PERSISTENT STATE	SIZE (RUNTIME BYTES)
Core (library)	nothing — pure and view functions: Ω, $\mathcal{D}$, Φ, Ψ, the Monoslot codec, safe transfers, `mulDiv`	pools and tokens, read-only, every guarded `staticcall` capped at `GAS_CAP = 100,000`	never — no `tstore`, no `sstore`	6,442
Hub	the registry (one Monoslot per pool, ≤ 16 per pair), factory rows (≤ 16), bridges (≤ 3), hook allow-list with codehash pins, V4 entries, roles; ERC-7201 namespaced	pools and factories, read-only, through the Core; the V4 PoolManager via `extsload`	on `recordSwap` (Router only), on admission and configuration	23,648
Solver	nothing — `view` only	the Hub (registry reads, discovery), pools through the Core	never	19,686
Router	admin, treasuries, `permit2`, `weth`, `paused`, `controlRenounced`, rescue queue; **no token balance at rest**	the input token, the pools of the route, the output token, the Hub (`isBridgeToken`, `isHookLive`, `recordSwap`), the Solver (`swapBestExactIn` only, `view`), WETH and the PoolManager for V4, Permit2 for its door, the treasuries, the recipient — never a caller-supplied target	its own configuration; the one external write per swap is `Hub.recordSwap` on the success path	23,781
Quoter	nothing	the Solver (`view`), the Hub (`isBridgeToken`, read-only), pools for the Exact Pass dry-run (called and reverted)	never	11,429

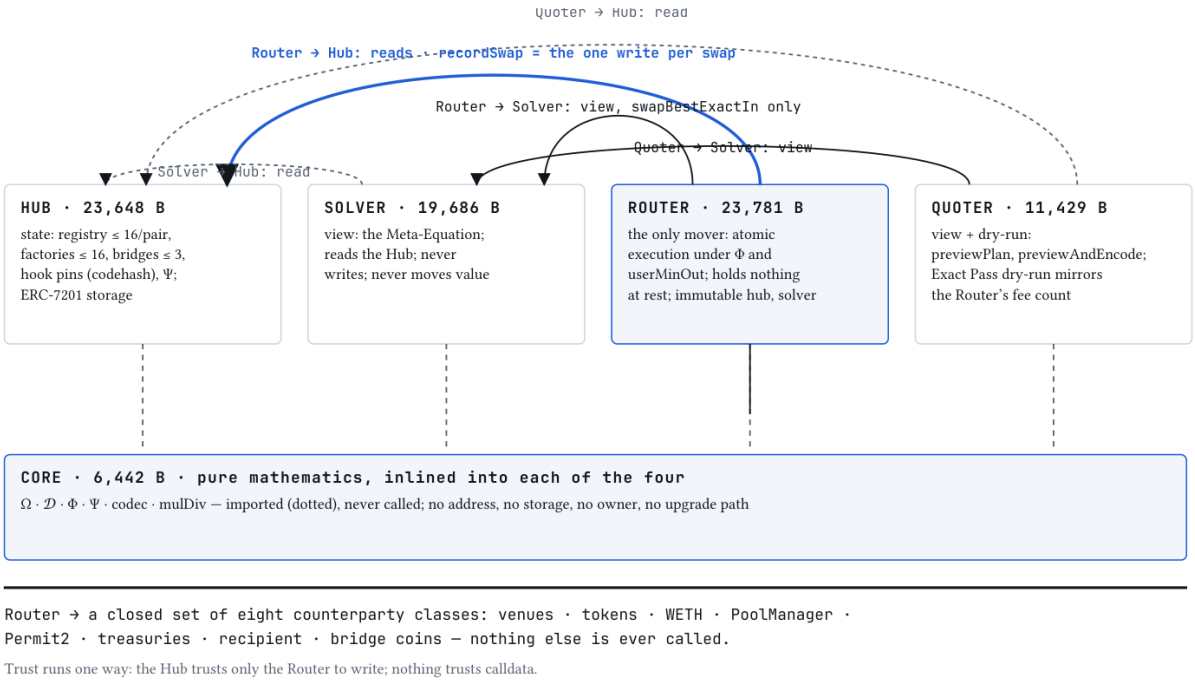


FIG. 15 Five-contract topology: the Core (pure mathematics: $\Omega \cdot \mathcal{D} \cdot \Phi \cdot \Psi$) imported by Hub (registry, Ψ state, bridges, hook pins), Solver (route optimiser, view), Router (atomic execution, the only contract that moves funds) and Quoter (preview, view and dry-run). Arrows are calls; the one solid arrow into storage per swap is Router → Hub.recordSwap; every other arrow from Solver and Quoter is view-only.

Trust runs one way. The Hub trusts the Router to call `recordSwap` (`onlyRouter`) and no one else; the Router trusts the Hub for three facts a route needs — is this token a bridge, is this hook live, and the registration write — and for nothing that prices a trade. The Solver trusts the Hub's registry only as a list of coordinates to measure. The Quoter trusts the Solver's plan as advisory and re-prices concentrated legs against the pools themselves. Nothing trusts calldata: a route reaches the Router as a stranger's claim.

§16.2 The life of one swap



FIG. 16 The life of one swap: `previewAndEncode` (view) → calldata → a door → `_execute` → venues → settlement, with the check at each step named by its refusal code: minimum and deadline at the door; hop shape and continuity; arrival measured; per hop, rescale, fee if this is the fee hop, per-leg floor and coverage gate, hop aggregate; sweeps; gross output; the route floor Φ; the output-side fee if any; delivery measured at the recipient; `userMinOut`; the fee ledger; `recordSwap` per executed leg; `ExecutionProof`.

1. **Preview** (off-chain, free): `previewAndEncode` runs `findBestRoutePlan` — registry and, if not fresh, discovery; Ξ; Ψ ranking; probes; the band; allocation; the clamp; the split gate; the attested floor — and returns the `Preview` and the calldata. The trader sets `userMinOut` from their own price expectation.
2. **The door**: `swapExactIn` (or a sibling) checks the deadline (`RouterE(4)`), refuses a zero minimum (`RouterE(10)`), pulls the input and measures the delta.
3. **`_execute`** : hop count, legs per hop and hop continuity (`RouterE(3)`); the fee scan; baselines born once.
4. **Per hop**: rescale against the measured arrival; charge the fee if this is the fee hop (or every hop in exhaustion); read each pool once for amount, impact and quote; execute each leg through the shape its kind selects (`RouterE(8)` for an unknown kind, `RouterE(9)` for a delta-altering hook); the per-leg floor with the coverage gate and the hop aggregate (`RouterE(5)`); the callback pays only the committed pool (`RouterE(6)`).
5. **Sweeps**: unspent input and bridge residuals above their baselines return to the payer.

6. **Settlement**: gross output as a delta (`RouterE(8)` if zero); `floorBps` from measured impact and concentration; the route floor against the final hop's in-frame quote and the three-way maximum (`RouterE(5)`); the output-side fee on a direct route into a bridge coin; transfer, delivery measured at the recipient, `userMinOut` on the delivered amount (`RouterE(5)`); the fee ledger (`RouterE(15)` , `RouterE(16)`).
7. **The one write**: `Hub.recordSwap` per executed leg, with measured amounts; then `ExecutionProof` , `Swap` and `Fee` events. Nothing is written when any check refuses; the transaction, every pool's swap included, never happened.

§16.3 Sizes against EIP-170

CONTRACT	RUNTIME BYTES	UNDER THE 24,000-BYTE GATE	UNDER EIP-170 (24,576)
Router	23,781	219	795
Hub	23,648	352	928
Solver	19,686	4,314	4,890
Quoter	11,429	12,571	13,147
Core	6,442	17,558	18,134
Total	84,986		

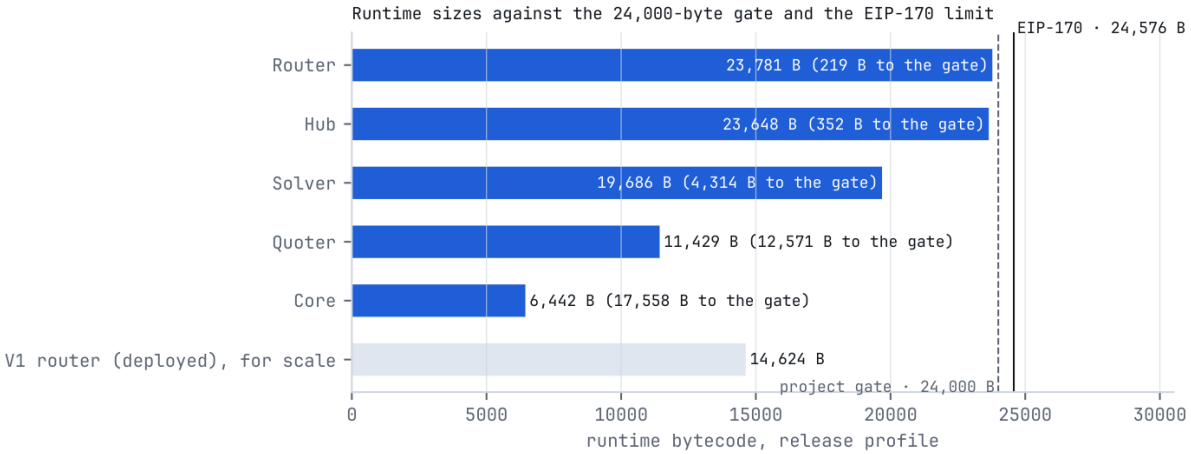


FIG. 17 Runtime sizes of the five contracts against the project's 24,000-byte gate and the EIP-170 limit of 24,576 bytes, measured on the release profile; the deployed V1 router (14,624 bytes) for scale.

The gate is asserted inside the suite itself (`DeployedSizeGate`) on the same profile the suite runs under, which `profile_parity.py` keeps identical to the release profile — a suite that tested a binary the chain could not accept would be testing the wrong object. Size is measured, never estimated: a guard estimated at tens of bytes once measured at several hundred because the shape chosen extended a variable's live range. The Router and the Hub sit within a few hundred bytes of the wall, which is why the pair-proof of §9 costs 179 bytes and the stable-curve repair of §6 is 35 bytes *smaller* than the code it replaced: every byte added is paid for by a byte removed.

§16.4 The attack surface, counted

An attack surface is what an adversary can write, what a contract holds, whom it calls, and who holds power over it. Each is a countable quantity here.

What a caller can write. A route is a typed structure — at most 3 hops × 5 legs, each leg ten fields (`pool`, `hooks`, `kind`, `fee`, `tickSpacing`, `zeroForOne`, `stable`, `amountIn`, `expectedOut`, `auxId`) — and no field names a call target or carries a calldata blob. `docs/assurance/fields.json` classifies every caller-writable field that reaches state, an event or a guard, 23 in all:

CLASS	COUNT	MEANING	EXAMPLES
confirmed	14	execution reads the field *and* a check compares it against an observation	`leg.kind` re-derived from the pool's shape (Hub `recordSwap`); `leg.fee` on concentrated pools, where the pool answers; `leg.amountIn`, rescaled by the measured arrival; `leg.expectedOut`, lifted by the coverage gate; `hop.tokenIn` / `tokenOut`, gated against the pool's own `token0` / `token1`; `leg.hooks` and `leg.auxId` on V4, which enter the executed pool id
steering	5	the field directs a token movement, so a false value punishes the caller and needs no separate check	`leg.pool` on pair and concentrated venues (tokens are pushed to it); `leg.zeroForOne` on V4 (a wrong direction fails closed in the manager's accounting); `route.singleOutFloor` (monotone — it can only tighten); the advisory route fields (`totalOut`, `estGas`, …), read by nobody
declared	4	it reaches state, an event or a guard, and nothing observes it — each with its reason recorded	`leg.fee` on V2 pairs (no pair exposes `fee()`; bounded by the 100 bps ceiling and used by quote and execution alike, so the two cannot diverge); `leg.hooks` on non-V4 legs (feeds only the hooked-last ordering; a lie costs the caller their own route); `leg.expectedOut` as published in the `Volume` event (bounded near +25 % by the per-leg floor that consumes the same field; no on-chain consumer); `leg.stable` (the Hub measures `stable()` at registration and both consumers read the same declaration)

The registry's ranking, the protocol floor and the fee base are computed only from confirmed quantities. The Hub confirms `kind`; the Router confirms the amounts, the tokens and the concentrated fees; the derivation confirms the V4 fields by construction.

What the Router holds between transactions. Zero token balance — `INV-1`, asserted by `invariant_RouterHoldsNothing`, `invariant_routerHoldsNothing` and `invariant_HoldsNothingBeyondTheSeed` across the campaigns, and watched by four guard mutants (a one-wei short transfer, and each of the three baselines removed). Fees stream to the treasuries inside the swap; residuals return to the payer inside the swap; the only thing that can sit in the Router is a mis-send, which the 48-hour rescue exists to return and which every swap's baselines exclude from payout. Three of the five contracts — Core, Solver, Quoter — emit no storage write at all.

Whom a swap calls. The counterparties of one swap form a closed set of eight classes: the input token; the output token; the pools the route names (at most 15); the Hub; the Solver, only through `swapBestExactIn` and only as a view; WETH and the PoolManager, only for V4 and native legs; Permit2, only through its door; the treasuries and the recipient. There is no ninth: the executor takes no caller-supplied call target (SOK-ARBITRARY-CALL), dispatching venue interaction from a closed set of shapes selected by a registry-held kind.

Every guarded read is capped at 100,000 gas. The exact instruction count of a swap is recorded rather than asserted: 4,382 Router instructions for one honest V2 swap on the release binary, with the opcode checked at every step (§15.10).

Who holds power. Nine control functions on the Router and six on the Hub, every one enumerated in §14.6–§14.7 with what it can and cannot reach; none can move principal, the fee, the split or the floors; all of them end at `renounceControl`, and what survives is the grow-only curator tier of four functions plus existing operator seats. There is no `selfdestruct`, no proxy, no initialiser after the Hub's one-time `initialize`, and the only `delegatecall`s in the artefact are the compiler's own into the public Core library.

What must exist for a trade to settle. §3.1's budget: no routing service, no oracle, no solver network, no keeper, no proxy — five zeros where the incumbent design has "required", "common" or "varies".

The counted difference. Against a design that fetches its route off-chain and executes a generic call list, or holds inventory:

SURFACE	THIS DESIGN	OFF-CHAIN-SOLVED EXECUTOR	INVENTORY-HOLDING VENUE OR ROUTER
caller-writable fields that reach a guard	23, of which 14 confirmed against an observation and 0 name a call target	a list of `(target, calldata)` pairs: unbounded, and each target is a call the contract makes on the caller's word	—
token balance at rest in the executor	0 (`INV-1`)	typically 0	the inventory itself
contracts that write storage during a swap	1 (`Hub.recordSwap`, one write per executed leg, success path only)	varies	the venue's accounting
who computes the route	the chain (`findBestRoutePlan`, `view`) or the caller, re-measured in-frame either way	a server; the contract checks, it does not choose	—
oracles, keepers, proxies, upgrade paths	0	commonly ≥ 1 of each	commonly a proxy
privileged functions that can reach principal	0	a proxy admin can replace the logic	varies
deployed bytes an auditor must read	84,986, five contracts, all under EIP-170	the settlement contract, plus a server whose size in auditable lines is unknowable	—

The comparison is counted, not argued: the difference is a bounded, typed, confirmed route against an unbounded list; zero against inventory; one write against many; and a route the chain derived — or the caller derived and the chain re-measured — against a route a server derived and the chain merely checked.

§17 Measured performance

Plain: everything in this section was measured on forks of real chains — real venues, real reserves deposited by strangers, one bytecode — and is labelled as exactly that. *Precise:* a number appears here only if a public harness in the repository reproduces it; the harness, the chain and the date are printed beside it. A fork measurement is evidence about behaviour, not a deployment record; the deployment record is §19. The measurements of §12 and §15 are from 2026-09-05; the campaign of this section was run in August 2026 and its harnesses are in `test/fork/`.

§17.1 Cross-chain identity

The same bytecode was exercised on forks of Ethereum, Arbitrum, Optimism and Base, adapting to each chain's venue mix — Aerodrome and Pancake alongside V4 on Base, Camelot on Arbitrum, Velodrome on Optimism — with no per-chain code change. The four chains were chosen because between them they exercise every venue family the registry admits, so a per-chain special case, had one existed, had four chances to surface. On every chain two identities were measured against ground truth: for CREATE2 derivation, each pool address computed by $\mathcal{D}$ against the address the chain's own factory reports for the same pair and parameters, with `hasCode` confirming bytecode at every derived address; for the Exact Pass, the venue's own swap called over live state and its deltas carried out in the revert payload, per chain, surviving each family's callback conventions, Algebra's included. Both identities hold on all four chains, and the differences column the table would need is empty.

§17.2 Preview → delivery fidelity across four chains

The thesis admits a direct measurement: quote a swap, execute it, compare. The `FidelityMatrix` fork harness quotes, executes and settles real pairs on Arbitrum, Base, Optimism and Robinhood Chain at pinned blocks and prints preview against delivery for every pair it touches; it carries a permanent alarm at 50 bps, so a fidelity regression announces itself. Measured 2026-08-24:

CHAIN	PREVIEW → DELIVERY	NOTED ALONGSIDE
Arbitrum	0–1 bps on every executed pair	the Solver's route matches an exhaustive two-way split grid in 10 % steps
Base	0–1 bps	within 0.13 bps of the grid maximum; real fills 222,000–326,000 gas all-in
Optimism	0–1 bps	the first fork-measured V4-routed execution by this engine on Optimism
Robinhood Chain	0–1 bps	grid-exact; the first measured executions on the chain

The matrix executes constant-product, concentrated, V4 — native included — and Solidly legs end to end. The band is one basis point on every executed pair, with a 50 bps alarm behind it; the Solver-versus-grid comparison of §17.1 is the one result that matches at the unit, and it is stated as such. Two behavioural results ride alongside: bridge intermediaries are selected adaptively per pair as the Meta-Equation's maximiser, and a deliberately isolated pair returns no route rather than a fabricated one.

§17.3 The factory census

A venue registration is a claim, and the claim has its own instrument. The census harness (`FactoryCensus`) inherits the production deployment wiring — it deploys through the same script the chains receive, behind an equality gate that refuses to run if harness and deployment drift. It lives with that script in the private deployment tree and is not in the public repository, so the table below is reported, not reproducible from this checkout; the public sweep that shares its fixtures and anti-vacuity gate is `test/fork/TokenSweep.t.sol`. It reports, for every registered factory on every chain over a fixed token corpus, three columns: registered, holds a pool for the pair, won the pair (supplied a leg of the winning route). The columns share one denominator, hop-pairs, with an explicit unresolved bucket that measured zero on all four chains: every leg the engine routed is attributed to a named factory.

ARBITRUM, 88 HOP-PAIRS	HOLDS A POOL	WON THE PAIR
Uniswap V4 (native + wrapped)	67	48
Uniswap V3	86	33
Pancake V3	77	13
Camelot V3	66	10

A pair credits every venue that supplied a leg of its winning route, so the column may exceed the pair count — attribution, not partition. On Arbitrum the V4 family wins more pairs than any other registered venue, the canonical V3 factory included — from-the-outside confirmation of the native-first probe order of §5.1. The census is also a pruning instrument: a factory that registers, holds pools and wins nothing over hundreds of measured pairs is a retirement candidate priced by attribution rather than reputation, and every sweep carries an anti-vacuity gate — a pair reported as having depth must also quote non-zero.

§17.4 Gas

Every figure is traceable to a named harness, cold and warm reported separately.

MEASUREMENT	VALUE	SOURCE
cold solve (first discovery, 4 factories)	169,093 gas	`test/fork/DiscoveryColdWarm.t.sol` ; `REPORTS.md`
warm solve (cached Monoslots, discovery skipped)	132,691 gas	same run
warm saving	−36,402 gas (−21 %)	difference
marginal cost per additional leg	≈ 28,700 gas	`LifecycleMetrics` (August 2026)
via-bridge second hop	≈ 59,000 gas	`LifecycleMetrics` (August 2026)
`previewPlan` on live Base, cold / warm / cold after TTL	1,524,221 / 1,414,633 / 1,359,925 gas	`test/fork/QuoterGasFork.t.sol` (§12.3)

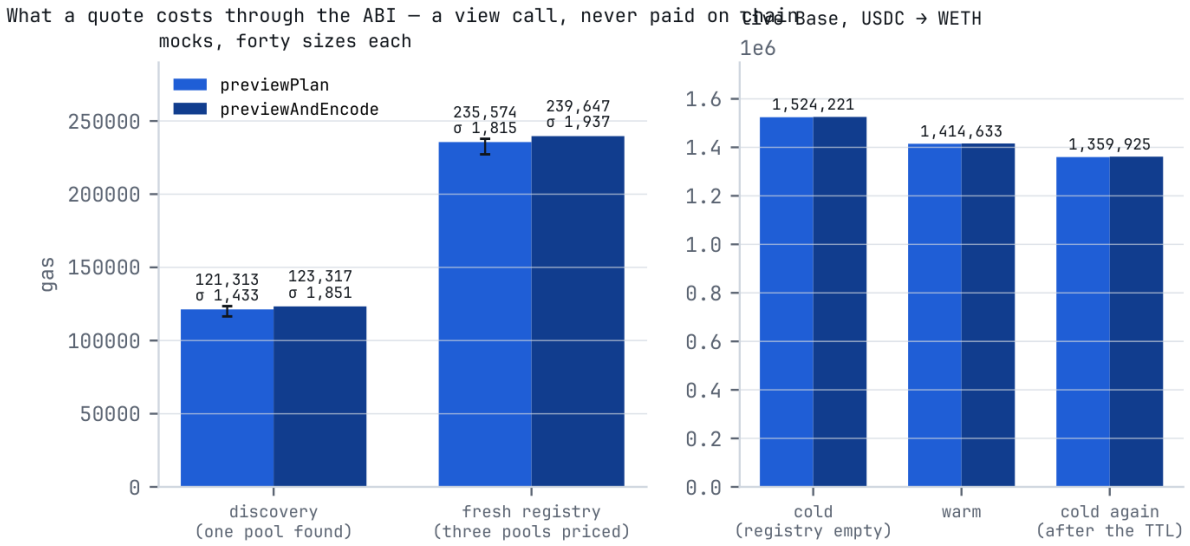


FIG. 18 What a quote costs through the ABI: `previewPlan` and `previewAndEncode` on mocks (discovery vs fresh registry, mean and spread over forty sizes) and on live Base (cold, warm, cold after the TTL).

Three structural readings hold without any measurement, because they follow from the architecture. The warm saving scales with the registry's breadth, not the trade's size: what the cold path pays for is sweeping factories, and the cached read that replaces the sweep is one `SLOAD` regardless. Legs are the unit of marginal cost, which is why the split gate exists (§10.4). And hops are the expensive dimension: a bridge route must win by enough at the pools to pay for itself at the pump. Callers choose their point on this surface per trade — solve in-frame and pay for the search, or supply a route and pay only for execution — and the freshness gate of §9.4 reserves the sweep for pairs the protocol does not yet know. The cost of a live quote is 1.4–1.5 million gas of *view* computation, paid by no one on chain.

§17.5 The executor against the field

A gas table without an external reference is self-praise, so the execution layer was benchmarked on the same fork, at the same block, with real production calldata: KyberSwap's `MetaAggregationRouterV2` (`0x6131B5fae19EA4f9D964eAc0408E4408b66337b5`), on a Base fork pinned to the block each route was built at, on 2026-08-22, with Paraswap quoted alongside as a second comparator: Kyber's own encoded routes, requested from its public API and replayed against its deployed contract, with our routes executed beside them and both sides' deliveries measured as balance deltas. The harness validates itself first: the competitor's measured delivery matches its API's promise to the wei, so a revert on either side cannot masquerade as a result. Per leg, the executor measured 2.4× to 10.9× more gas-efficient than the reference — on the clearest pair 322,172 gas for four legs against 879,553 for one, 2.7× per transaction — by a harness (`BenchExec.t.sol`) that lives with the captured calldata in the private tree, so the numbers are reported, not reproducible from this checkout, which is compiled at `optimizer_runs = 999999` — so the difference is design, not compiler settings; their router interprets a generic call list a server decided, ours executes a typed structure, and interpretation costs. On the clearest pair their single-leg route cost 879,553 gas; four of our legs cost 322,172. Across nine measured pairs the fully solving door and the calldata door delivered identical output. The method's error bars are part of the result: quotes were pinned to the block they were built for, and API-side gas estimates were discarded after real execution measured one of them 148 % low.

§17.6 The byte–gas exchange rate ρ*

Which door should a caller use — carry a route in calldata, or let the chain solve? A transaction's cost is affine and separable — $21{,}000 + \text{bytes} \cdot p_{\text{byte}} + \text{gas} \cdot p_{\text{gas}}$ — so the whole trade between calldata and execution reduces to one scalar: **ρ***, the execution gas one byte of calldata buys. Measured with `test/EquationBench.t.sol` on the release profile, ρ* runs 66.9 gas per byte on single-leg routes to 105.5 on four- and five-leg routes, whole transaction. Against it stands each chain's live byte price, read from the chain's own fee predeploys — the OP-Stack gas-price oracle, Arbitrum's `ArbGasInfo` — with no key and no third-party feed.

CHAIN, PRICES READ LIVE 2026-08	p_{byte} (GAS/BYTE)	DATA SHARE OF A SWAP	CHEAP DOOR
Base	7.2	0.9 %	calldata route
Arbitrum	12.8	3.4 %	calldata route
Optimism	33.9	14.0 %	calldata route
ZK rollup paying calldata-priced data availability	243,256	99.9 %	in-frame solve — 14× cheaper end to end

Where $p_{\text{byte}} < \rho^*$ the calldata door is the cheap one; where $p_{\text{byte}} > \rho^*$ the in-frame solve is. On the OP-Stack and Arbitrum chains the data term is 0.9–14 % of a swap's cost, so execution is where the money goes — which is where the executor efficiency above compounds. The Router's four doors are not a convenience feature; they are the affine cost model, made into an interface.

§18 The DEX, stripped to the bone

Seventeen sections of mechanism reduce to one page. If you read nothing else, read this.

One job. Take two tokens, an amount, your minimum, a recipient and a deadline. Return the most output the public liquidity of the chain can deliver — or refuse, with a code that says why. There is no third outcome: in the hostile-venue matrix, twenty of twenty venue-pathology-and-door fixtures settle inside the floors or refuse with a selector of ours (§15.5).

Five contracts, one mathematics.

CONTRACT	WHAT IT IS, IN ONE LINE
Core	The stateless library: every venue's closed form, the derivation $\mathcal{D}$, the floor Φ, the vitality score Ψ. Defined once, imported by the other four.
Hub	The registry that learns liquidity by trading it — and proves every pair before it stores it. It prices nothing.
Solver	Builds the route and the split from *measured* marginal output and depth. `view` only.
Router	The only contract that moves funds. Measures every balance delta, enforces the floor, holds nothing at rest.
Quoter	The free preview — the same evaluator the Router settles with, so preview and delivery agree.

Six venue kinds, four families. Uniswap V2, V3 and V4 (native ETH included), Algebra dynamic-fee pools and Solidly stable and volatile pairs — each priced by one closed form in the Core, one dispatcher Ω for all of them (§4). Curve and Balancer were excised, and their kind numbers retired for ever (§4.1).

Four doors, one executor. `swapExactIn`, `swapExactInWithPermit2` and `swapExactInNative` take a route in calldata; `swapBestExactIn` solves and executes atomically on chain. All four reach the same `_execute`, the same floors and the same fee (§11.1).

Three floors under every trade. A per-leg floor, a hop aggregate, and the route floor Φ re-derived from measured impact in the executing frame — then *your* `userMinOut` on what actually arrives at the recipient. Delivery is the larger of the protocol's floor and yours (§11.5).

One fee, compiled in. 0.28 %, charged once on the first bridge coin the Router holds, rounded up so it is never zero, and nothing that grows with the gap between quote and delivery (§13).

What it refuses to trust. Advertised liquidity. Declared fees. Calldata depth. A pool's own account of which tokens it trades. Every one is either measured from the pool or derived by arithmetic before it can reach shared state (§7, §9, §16.4).

What it costs. Gas: deriving the route in-frame is dearer than settling an answer computed elsewhere, and §17.6 tells you per chain which door is cheapest. Immutability: a defect is disclosed and bountied, never patched (§21).

§19 Deployment status

This paper documents the repository, and the repository is canonical: where this text and the source disagree, the source wins. Two generations exist and they are never conflated.

V1 is what is live on chain. It is a different, earlier codebase, archived in its own repository — not an earlier commit of this one. Measured by RPC on 2026-09-04, the deployed V1 router is the same 14,624-byte contract on Base, Ethereum, Optimism and Arbitrum. `test/fork/DeployedParity.t.sol` and `test/fork/DeployedCodehashPin.t.sol` pin what V1 is, on every network the SDK names — the codehash of each deployed contract, recorded in `test/fork/deployed-codehash.json` — so a change to what is deployed is noticed, and they assert the properties that hold regardless of generation: no `SELFDESTRUCT`, no `CALLCODE`, no `tx.origin` authority, a transient reentrancy lock, a fixed runtime size, live wiring, and control not yet renounced.

This repository is V2, and V2 is not deployed. The tree measured in this paper builds a 23,781-byte Router and carries external functions V1 does not have — among them the native-value door, the wrap/unwrap seam, the rescue path and the Permit2 door. The V2 contracts are deployed onto forks of live liquidity and exercised end to end on every fork run (§15.1, §17). Nothing in §15 or §17 protects the deployed V1 contracts; every claim in this paper is a claim about V2 as it stands at `8949a9d`.

Two policies protect readers accordingly. This paper contains no contract addresses — addresses outlive the documents that quote them, and the live address surface is maintained where it can be updated and machine-checked: the repository's records and the site's verification page and manifest. And a deployment record has a

specified format — chain, address, deploying commit and runtime-bytecode hash, written at broadcast time, with a verification step that re-derives the hash from a clean build — so that when V2 deploys, every address claim is arithmetic, not trust.

§20 Token — BZPX

One billion BZPX, fixed for ever, as a plain ERC-20: 18 decimals, no mint path, no fee-on-transfer, no rebasing, no transfer hooks, no pause or blacklist — six deploy invariants, checked against the token bytecode at wiring time. Fully-diluted supply is known from genesis; scarcity is a property of the contract, not a policy.

ALLOCATION	TOKENS	SHARE	NOTE
Market liquidity	520,000,000	52.0 %	the float the aggregator routes through
Staking emission	180,000,000	18.0 %	two independent staking deployments, each capped at 90,000,000 over 8 years: a linear base plus a bonus that halves every two years (corrected in 2.3.1)
Airdrop & Bounties	130,000,000	13.0 %	one pool: the former 50,000,000 airdrop, the former 50,000,000 research pool of §21, and 30,000,000 moved out of market liquidity
Team and operations	103,000,000	10.3 %	the only locked insider tranche; 30-month linear vesting
Partners and ecosystem	60,000,000	6.0 %	integrations, venue partnerships
Marketing	7,000,000	0.7 %	launch awareness

What changed in September 2026. The airdrop and the security-research pool were two lines paying the same people for the same thing — reading the protocol hard and using it for real. They are now one line of 130,000,000 BZPX: 50,000,000 from the airdrop, 50,000,000 from the research pool, and 30,000,000 taken from market liquidity, which moves from 550,000,000 to 520,000,000. Nothing was taken from anyone else's line, and the supply is still exactly one billion.

The token contract's on-chain record, once published, is the canonical version of this table. At the time of writing the token has no provisioned liquidity and no token generation event; on-chain volume of zero is the expected state of this stage, not a signal about the code.

§21 Research programme and security record

Before this project had a research programme worth the name, independent researchers read the source hostilely — unasked, with nothing promised — and disclosed what they saw privately. The code this paper describes is the code that survived their reading. What each of them reported, and what became of it, lives where such records belong: in the repository's `SECURITY.md` and `SECURITY_HALL_OF_FAME.md`, next to the code, where every entry can be checked in both directions. This section records who they are, what hostile reading pays, and the shape of what it found.

The researchers. NetGakarot · duxun · AmanDara1 · amitbhakar · auditor_1b3f2c · siam siddik · Thomas · llen · destinyae · superagent · Mohd Huzaifa · Raditya · bai bo · Josh W · Borutobro · mohaseenkatika · mohaseenbasha · Karan Rathod · Seavia Resources · acit aja · Brian Wahyu · M2000 Slash · Binod Bk · CHARIR ABDELHAMID · and one researcher who chose to remain anonymous. Twenty-five researchers have been credited; no report has reached the Critical class. Attribution is never rounded down: names are published exactly as given.

The pool: Airdrop & Bounties, 130,000,000 BZPX — 13 % of the fixed supply (§20). Security awards are drawn from it at the bands below, which are unchanged.

SEVERITY	AWARD
Critical — direct theft or permanent freezing of user funds; a reachable insolvent state	2,500,000 – 7,500,000 BZPX
High — theft under specific conditions; a redistribution that systematically pays the wrong party	625,000 – 2,500,000 BZPX
Medium — griefing, temporary denial of service, recoverable residue	125,000 – 625,000 BZPX
Low — demonstrated impact below the above	up to 125,000 BZPX

Awards are paid in BZPX — quantity and schedule promised, market value not — with payouts beginning after October 2026 and reports triaged now. One clause is deliberate: a report need not break a conservation law to be Critical — a redistribution that systematically pays the wrong party while the books balance is priced like a theft, because that is the class only hostile readers catch.

What the record contains, by class. Every confirmed finding became a named property, a regression test that fails against the pre-fix code, and a mutant the test must kill; the classes, one sentence each, by reference to the register:

- *Two producers of one quantity* — a planner and an executor, a preview and a delivery, answering one question in two frames (the floor's impact aggregation, the bridge named by the preview against the hop the fee anchors on, the leg budget across two contracts, the frozen bridge bit against the live predicate): closed by collapsing to one producer or by a named parity test, and enumerated as a register with a CI rule.
- *A caller-supplied quantity where a measurement existed* — the fee base read from `route.totalOut`, the registry's fee and hooks written from calldata, depth declared rather than measured, a published volume that reported the declaration: closed by measuring, and by the field classification of §16.4.
- *Absence read as permission* — a final hop that could not be quoted in-frame leaving the floor inert, a coverage gate a caller could switch off by writing zero, an empty fee list that quoted a whole family at zero: closed by falling back to the attested or measured figure, never to nothing.
- *A pin on the wrong object* — a factory's derivation origin that a proxy could move while its codehash stayed, a hook's runtime pinned where its implementation could move: closed by attesting the answer where the code cannot be pinned, and by stating the residual at the site where it cannot be closed.
- *Fail-open in a quote* — the stable curve's domain guard that itself overflowed on dust, found by the N-version lane: closed red-first with the repair smaller than the code it replaced.
- *A fix applied to one of two symmetric channels* — the defect signature this codebase names first, met repeatedly; every closure now searches for the sibling before it merges.

The invitation is the point of the section, and it is standing: independent readers are invited, in public, and paid, in a currency whose quantity is fixed. Report privately, through the disclosure channel in the repository's `SECURITY.md`.

§22 What you must trust, and what you need not

Every protocol claims to be trustless; almost none can enumerate its exceptions. Here is the full list, each power stated with the line that bounds it.

- **The venue registry, the bridge list and the V4 hook allow-list** — curated; a hostile listing's worst case is bounded by the stricter of your minimum and the Iron-Law floor Φ; principal it cannot reach (§9.6, §14.6).
- **The Router's control key, before renunciation** — can redirect the fee destination, pause swaps, repoint the WETH and Permit2 contracts its native and Permit2 doors call, and recover mis-sent tokens behind a 48-hour public delay; cannot move the fee rate, the split or the floors (compile-time constants, no setters) or reach principal (no proxy, no upgrade path, no `selfdestruct`, nothing held at rest).
- **The Hub's control key, before renunciation** — can repoint the Hub's view of the Router, Solver and Quoter, repoint the V4 manager, pause the registry's learning, and grant operator seats; cannot misprice a fill, because nothing that prices a trade reads the registry.
- **The treasuries** — fixed at construction, and the 30 / 70 split has no setter.
- **The canonical contracts the doors depend on** — Uniswap's Permit2 for the Permit2 door, the chain's WETH for the native door, the PoolManager for V4 legs: each is called only through its door or its leg, and the Router prices every leg off its own measured balance delta, so a dishonest counterparty collapses the output and your bound refuses the swap.
- **The token and the correspondence** — that BZPX satisfies its six deploy invariants, and that the addresses you call correspond to this text: checkable the moment §19's records publish, by the procedure §19 specifies.

What you need not trust: anyone to price your trade — the route is derived in the executing frame from public state, or re-measured there if you supplied it; anyone to route it fairly — the floor is re-derived on-chain from measured output and the fee has no term that grows with the gap between quote and delivery; anyone to count the fee — the Router refuses to settle without paying it; and anyone to keep serving you — if every server this project operates disappeared tonight, the contracts would quote, route and settle tomorrow exactly as they did today.

And if you keep one habit from this document, make it the minimum you set on your own trades. Every bound in the aggregator falls back to that number; it is the only protection in the system whose quality depends on nobody's competence but yours.

§23 Related work

Routing. Angeris, Evans, Chitra and Boyd cast routing across constant-function market makers as a convex program whose optimum equalises marginal prices across venues; Diamandis, Resnick, Chitra and Angeris give an efficient dual-decomposition algorithm for it. This design sits deliberately short of that optimum on chain: the Split Allocator is proportional-to-depth in one pass (§10.4), which needs no iteration and no per-venue curvature model and is robust where an iterative solver is a gas bill and an attack surface; the metamorphic lane measures the distance the choice costs — the split never pays less than the best single pool (MR-R1) and the fidelity matrix places the Solver's route within 0.13 bps of an exhaustive grid on Base. The exact marginal-return-equalising closed form for the constant-product family is two sums and a square root per venue and is a candidate for a later release; the objective is flat near its optimum, which is why the robust rule ships first. Off-chain solver designs move the optimisation to a server and reduce the contract to a checker; the Dependency Budget of §3.1 is this paper's statement of what that costs. Recent work extends the model with gas: *Optimal Routing across Constant Function Market Makers with Gas Fees* shows gas widening no-trade regions and interacting non-trivially with fragmentation, and *Routing Frictions and Executable Liquidity in Fragmented Markets* separates advertised from executable liquidity — the distinction this design measures in-frame rather than models.

Robust aggregation. The believability centre of §10.3 is a depth-weighted median, whose breakdown point is half the weight. Müller, Moumeni and Messaoudi, *Cost of Manipulation in AMM-Based Oracles* (FC 2026), prove the corresponding bound for oracles that aggregate AMM prices: with equal weights the cheapest attack moves the shallowest pools, while with liquidity weights controlling half the weight requires at least half the total quote depth. Ormer proposes a median-estimating on-chain oracle in place of the TWAP mean, and *From Means to Medians* studies which benchmarks are hardest to manipulate. E25 applies the same principle to routing: a pool is believed only as far as capital at risk stands behind it.

Venues. Uniswap v2 gives the constant-product reference and the 30 bps convention the V2 branch defaults to; Uniswap v3 the concentrated-liquidity model, `slot0` and Q64.96 fixed point the closed form of §4.2 follows; Uniswap v4 the singleton, flash accounting and the hook-permission address encoding that §5 and §14.2 rest on — the design here is, to our knowledge, the first to derive, prove, price, screen and settle V4 pools entirely in-frame. Egorov's StableSwap defines the amplified invariant of the Curve family this protocol excised rather than support on an unmeasured trust assumption (§4.1); the Solidly stable curve of §6 has no peer-reviewed paper and is cited by its deployed implementations.

MEV. Daian et al. named the ordering adversary and showed its scale; Zhou et al. measured sandwich attacks on decentralised exchanges and formalised their profitability. This paper claims no elimination — ordering is decided before contract code runs — and instead measures the bound: the sandwich curve of §14.3, from the attacker's side, with the refusal edge and the attacker's negative round trip past it.

Verification. Clarkson and Schneider's hyperproperties name the class the redistribution findings of §21 belong to — properties of sets of traces, not of one trace — and why a conservation law can hold while the wrong party is paid. Cousot and Cousot's abstract interpretation is the frame for the single-tick closed form: an abstraction of the venue's mathematics that is exact within a range and sound in one direction outside it, with the floor as the concretisation check. Chen, Cheung and Yiu introduced metamorphic testing, the second judge of §15.7 where no oracle independent of the formula exists. DeMillo, Lipton and Sayward introduced mutation analysis, whose coupling hypothesis the curated guard of §15.2 relies on. Avizienis's N-ver-

sion programming is the ancestor of §15.8, applied to code generation rather than to independent teams. Kuhn, Kacker and Lei's combinatorial testing gives the covering arrays of §15.4 their construction and their claim: every t-way interaction of regime factors exercised by construction, with the denominator printed. Littlewood and Strigini's result on validating ultra-high dependability is why no probability of correctness is stated anywhere in this paper.

Standards. EIP-1153 transient storage carries the One-Callback Doctrine's leg context, the fee ledger and the reentrancy lock; EIP-170 is the 24,576-byte wall the Router and Hub sit within a few hundred bytes of; EIP-1014 is the CREATE2 arithmetic of $\mathcal{D}$; EIP-2612 and Uniswap's Permit2 define the signature-based transfer of the second door; ERC-7201 namespaces the Hub's storage; EIP-7702 delegated accounts call the same doors as any contract.

§24 Conclusion

We set out to build an aggregator on one discipline — measure, do not believe — and to hold it at every point where a number could instead have been taken on someone's word: the route, the price, the floor, the fee base, the venue set, the pair a pool claims to trade, even the protocol's own cost model.

The result is a small system with a strong shape. Liquidity fragmentation turned out to be 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 — a dispatcher, a derivation, a floor, a vitality field — can price the landscape without per-venue special-casing, derive the route inside the transaction that executes it, and re-check its own promise against measured output before a token reaches you. The measured record is the shortest summary of what that buys: a drift-free quote that settles at exactly its prediction 64 times in 64, on mocks and on live Base; a floor that caps a sandwich on a 1 % trade at about 2.7 % of it and refuses beyond; a preview and a delivery separated by 0–1 basis points across four chains; and an apparatus that publishes, with denominators, how it would be gamed and what it cannot see.

What it costs is equally plain. Gas: deriving a route in-frame is dearer than settling an answer computed elsewhere — the price of a guarantee you can check instead of one you must take on someone's word — and the byte–gas exchange rate tells you, per chain, which door makes that price smallest. Immutability: defects are disclosed and bountied, not patched — the price of a contract you audit once, and §21 is what that price buys.

The thesis, restated: the thing that chooses is the thing that trades. A quote and its execution are the same computation, so the gap where an operator's honesty used to live is closed by arithmetic; a registry that learns liquidity by trading it, and proves every pair it stores; a fee, a floor and an entry surface that nobody — the authors included — can move, because they are compiled in. The protocol does not ask to be trusted. It asks to be verified, and it works to make verification cheap, public and permanent.

Revision history

Version 2.3.1 (2026-11-16)

Corrections only, against version 2.3: the code licence's Change Date (2030-06-01, not 2030-07-01); the staking line of the §20 allocation table (two deployments of 90,000,000 BZPX each over 8 years, a linear base plus a biennial-halving bonus, as the staking contract at `c0366cf` sets it); the text licence of this version (CC BY-NC-ND 4.0); and a dated status line. The anchor stays `main` at `8916545`.

Version 2.3 (2026-09-24)

Version 2.3 continues the 2.2 line; the 3.0 numbering used briefly for a draft of this text is withdrawn, as the 3.0 numbering of 2026-09-01 was before it. What changed against 2.2: the eighteen innovations are named, numbered I-1 to I-18 and stated with their mathematics at the front of the paper, with a scoreboard of the measured record; every figure is re-measured against `main` at `8916545`; the staking engine is no longer described here (its own specification carries it); the token section records the single Airdrop & Bounties pool of 130,000,000 BZPX; and the errata of 2.2, below, are folded into the text they corrected.

Errata to 2.2 (2026-09-07)

Applied after a hostile read of the 2026-09-06 text against `main 8949a9d`; the version line stayed 2.2. Each item names what the text said and what the checkout says.

1. §4.2 — the boundary clamp `sqrtBoundary` is defined in the Core but not yet wired into ranking or the in-frame promise; the text said it was applied on the promise side. Open item: wire it (`Router._v4LegQuote`, Core ranking arm), with a red-first test that delivers below the net quote beyond the E21 buffer when a V4 leg leaves its range.
2. §10.4 — the capacity clamp excludes V4 legs (the Solver skips the balance read for kinds 4 and 8); the text said every concentrated leg was clamped. Open item: a physical bound for the singleton.
3. §5.4 — a static-fee V4 key ignores the manager's protocol fee (zero on every chain served today); the text called the key fee unambiguous.
4. §9.3 — the eviction margin runs down to about 25 % (incumbent 51, newcomer 64), not 3 %; the earlier figure violated E13 and was copied from a source comment.
5. §15 and the abstract — 43 `invariant_*` functions (the printed command returns 43), 217 `.t.sol` files.
6. §14.1 — `HubE(6)` does not exist; factory exhaustion is `HubE(4)`, slot exhaustion is `_canInsert`.
7. §14.2 — "any one layer suffices" withdrawn: the layers bound delta alteration; the fee-override residual of §5.4 is bounded by the floor.
8. §11.4, INV-21 — the route-shape rule is canonical order route-wide (`sawHooked`), not "hooked only in the last hop".
9. §11.3, E24 — the aggregate hop floor's slack is divided by BPS: 20 % of the average attested leg.
10. §17.3 and Appendix C — the `FactoryCensus` harness lives in the private deployment tree; the table is reported, not reproducible from this checkout.

11. §17.5 — the gas reference is named (KyberSwap `MetaAggregationRouterV2`, Base fork, 2026-08-22, Paraswap as second comparator) and the per-transaction ratio (2.7×) is printed beside the per-leg one; the harness and captured calldata are private, so the numbers are reported, not reproducible from this checkout.
12. §17.2 — "the chain's first V4-routed aggregation" withdrawn; "exact" replaced by the measured band (0–1 bps).
13. INV-8, INV-10, INV-15, INV-20, INV-22 — enforcement citations corrected to the tests and codes that carry them; INV-22 names the four deployed contracts.
14. §1 — the survey of aggregators is bounded to the seven named, as documented on 2026-09-07.
15. References 16 and 20 — EIP-1153 is by A. Akhunov and M. Salem; EIP-7702 by V. Buterin, S. Wilson, A. Dietrichs and M. Garnett.
16. §2.3, §15.11, §21 — instrument-calibration and defect-pattern counts replaced by the calibration rule itself.

- §14.2, Layer 2 (2026-09-07, hook admission): the allow-list curates the automatic door and the code-hash pin pauses a listed hook whose code moved, on every door. A hook whose address bits carry no swap permission never runs in a swap and needs neither list nor pin. A hook that runs in the swap and that nobody listed is accepted on the explicit door as the caller's signed choice, bounded like every venue by the in-frame promise, the attestation gate and `userMinOut`; it is never proposed by the automatic door. Registering a hooked pool admits and pins its hook, so a hooked pool is one operator step. The three layers stand; "default closed" describes the automatic door.

Errata to 2.2 (2026-09-24)

Applied after the ninth bounty wave; the version line stayed 2.2.

1. §4.2, and item 1 of the 2026-09-07 errata — closed. A V4 pool is quoted by walking its own book: its price, liquidity, tick bitmap and ticks, read through `extsload`, with the PoolManager's own step arithmetic, for ranking and for the promise alike (`Core.v4WalkOut`). The boundary clamp `sqrtBoundary` is retired. The statement that the dispatcher does not replicate tick-crossing arithmetic now holds for the V3 family only.
2. §10.4, and item 2 of the 2026-09-07 errata — closed for the purpose it served: the walk counts only the liquidity the book holds in the ticks it crosses, so a V4 leg is bounded by what its own pool holds (`test_AOneTickRangeWithNothingBeyondPaysWhatItHolds`).
3. Item 3 of the 2026-09-07 errata — closed: a static-fee key composes the manager's protocol fee with the key's LP fee in the swap's direction (`Core.effV4Fee`; `test_StaticKey_ProtocolFee_ReachesThePromise`).
4. §4 — every read the quote path makes of a pool is gas-capped and copies a fixed number of words; the two that were high-level calls (the Solidly pair's `getAmountOut`, and `factory()` then `getFee`) ask through `Core._askWord` (`test/APoolsAnswerIsBoundedByTheReader.t.sol`).

Appendix A · The primitive equations

Every quantitative claim in this paper reduces to one of the following. Each is implemented once, in the contract named, and imported everywhere else. The numbering E1–E22 is stable from the previous edition; E23–E26 are new and appended.

#	NAME	EQUATION	DEFINED IN	§
E1	Meta-Equation	$R^\star = \arg\max_R \ \mathrm{net}(R) \cdot \prod_\ell \hat{\Psi}(p_\ell) \cdot \mathbf{1}[\mathrm{net}(R) \geq \Phi(R)] \cdot \Xi(R)$	Solver pipeline	§3
E2	Constant-product output	$y = \lfloor x(\mathrm{BPS} - f)\, r_{\mathrm{out}} / (r_{\mathrm{in}}\,\mathrm{BPS} + x(\mathrm{BPS} - f)) \rfloor$	Core `outV2`	§4.2
E3	Concentrated-liquidity output (within tick)	$\sqrt{P}' = L\sqrt{P}/(L + x_f\sqrt{P}/Q_{96})$, $y = L(\sqrt{P} - \sqrt{P}')/Q_{96}$; dual form for the other direction	Core `outV3`	§4.2
E4	V4 storage base slot	$\mathrm{slot} = \mathrm{keccak256}(\mathrm{abi.encode}(\mathrm{poolId}, 6))$; $\sqrt{P}$ in bits [0,160), LP fee in [208,232) of word 0; L in word 3	Core `v4SqrtAndLiq`	§5
E5	Solidly stable invariant	$k(x,y) = x^3y + xy^3$	Core `_solK`	§6
E6	Solidly Newton step	seed $y_0 = Y$; up-step $\max(1, \lfloor (K-k)/f' \rfloor)$; down-step $\max(1, \lfloor (k-K)/f' \rfloor)$, replaced by $\lfloor y/2 \rfloor$ when it would cross zero; exit at $\lvert y_{n+1} - y_n \rvert \leq 1$ with a one-unit bump against the taker; cap 64 → seed → quote 0	Core `_solY`	§6
E7	Decimal normalisation	$X = r_{\mathrm{in}} 10^{18-d_{\mathrm{in}}}$, $Y = r_{\mathrm{out}} 10^{18-d_{\mathrm{out}}}$, de-scaled after the solve	Core `_solidlyStable`	§6
E8	CREATE2 derivation	$\mathrm{pool} = \mathrm{addr}_{20}(\mathrm{keccak256}(\texttt{0xff} \Vert \mathrm{origin} \Vert \mathrm{salt} \Vert \mathrm{initCodeHash}))$	Core `create2Address`, `deriveAddress`	§7
E9	Monoslot packing	$s = a \mid f \ll 8 \mid k \ll 32 \mid r \ll 40 \mid (c \wedge \texttt{0xFFF}) \ll 48 \mid b \ll 60 \mid \ldots \mid B_{\mathrm{last}} \ll 224$	Core `encodeSlot`	§8
E10	Vitality Field	$\Psi = V \cdot 2^b \cdot (1 + 0.25\,\mathbf{1}_{\mathrm{bridge}}) \cdot (1 + 0.05\,\mathbf{1}_{\mathrm{conc}})$	Core `psi`	§9.2
E11	Activity decay	$V = \mathrm{swapCount} \gg \lfloor (t - t_{\mathrm{last}})/24{,}576 \rfloor$; 0 once the shift exceeds 31	Core `_decayedSwapCount`	§9.2
E12	Depth bucket	$b = \min(15, \lfloor \log_{10}(d/10^{15}) \rfloor)$	Core `depthBucket`	§9.2
E13	Eviction hurdle	$2^{b(d_{\mathrm{new}})} > \Psi_{\mathrm{worst}} + \lfloor \Psi_{\mathrm{worst}}/4 \rfloor$	Hub `_canInsert`	§9.3
E14	Split gate	keep the split iff $\mathrm{out}_{\mathrm{split}} \geq \mathrm{out}_{\mathrm{single}} \cdot (1 + 25/10^6)$, $\mathrm{out}_{\mathrm{single}}$ the better of the deepest and the best-rate candidate at full size	Solver `_buildHop`	§10.4
E15	Iron-Law floor	$\texttt{floorBps} = \max(9{,}600 - \texttt{legShv} - \min(\delta, 10^4) - \sigma_{\ln}/10^{14},\ 8{,}000)$	Core `ironFloorBpsShv`	§11.5
E16	Effective minimum	$\texttt{effMin} = \max(\texttt{userMinOut},\ \texttt{route.singleOutFloor},\ \lceil \texttt{finalHopQuote} \cdot \texttt{floorBps}/10^4 \rceil)$; the second term dropped and the third re-scaled by the measured net ratio when fee-on-transfer is detected	Router `_execute`	§11.5

#	NAME	EQUATION	DEFINED IN	§
E17	Protocol fee	$\text{fee} = \lceil 28 \cdot \text{base}/10^4 \rceil$, base = the fee hop's measured input (anchored), the gross output (direct route into a bridge coin), or each hop's measured input (exhaustion); $t_1 = \lfloor 3{,}000 \cdot \text{fee}/10^4 \rfloor$, $t_2 = \text{fee} - t_1$	Router `_chargeHopFee`, `_payFee`	§13
E18	Hook gate	$(\text{uint160}(\text{hook}) \,\&\, \texttt{0x3FFF}) \,\&\, ((1 \ll 3) \mid (1 \ll 2)) \neq 0 \Rightarrow$ leg refused, no call made	Core `hookAltersDeltas`; Router	§14.2
E19	Dynamic-fee gate	$f_{\text{eff}} = f_{\text{key}}$ if static; unquotable if the sentinel rides with a non-zero protocol fee; else the live LP fee from `slot0` / `globalState` / the pool's `fee()`	Core `effV4Fee`, `quoteV3Fee`	§5.4
E20	V2 fee ceiling	$f_{\text{eff}} = f_{\text{decl}}$ if $0 < f_{\text{decl}} \leq 100$ bps, else 30	Core `effV2Fee`; the same rule for the Solidly declared fallback	§4.2
E21	Preview safety buffer	$s = \min(\max(0, n-2) + 5a,\ 10)$ bps	Quoter `_pack`	§12.1
E22	Surface choice	carry the route in calldata iff $p_{\text{byte}} < \rho^*$; ρ^* measured 66.9–105.5 gas/byte by route shape	`test/EquationBench.t.sol`	§17.6
E23	Leg shave from concentration	$\texttt{legShv} = \sum_{\text{hops}} \lfloor 200\,((\sum a)^2 - \sum a^2)/\sum a^2 \rfloor$	Core `legShaveBps`	§11.5
E24	Aggregate hop floor	$\sum \text{got} + \lfloor (\text{BPS} - 8{,}000) \cdot \lfloor \sum \text{attested}/n \rfloor / \text{BPS} \rfloor \geq \sum \text{attested}$	Router `_execute`	§11.3
E25	Believability centre	the depth-weighted median: sort (r_i, d_i) by r_i, return the first r_i at which $\sum_{j \leq i} d_j \geq \lceil \sum d/2 \rceil$; band $[0.95, 1.05]\times$ centre	Solver `_depthWeightedMedian`	§10.3
E26	Fee regime predicate	$\text{feeHop} = \min\{h : \text{isBridge}(\text{hops}[h].\text{tokenIn})\}$; charge at h iff $\neg\text{feeOnOut} \wedge (\text{feeHop} = \infty \vee h = \text{feeHop})$; $\text{feeOnOut} \iff$ one hop and $\text{isBridge}(\text{tokenOut})$	Router `_execute`	§13.1

Appendix B · The invariants

These are the properties that hold in every reachable state: each is enforced by the shape of the code at the site named, and a transaction that would violate one does not execute. INV-1–INV-18 keep their numbers from the previous edition, with the enforcement column brought to the code; INV-19–INV-22 are new.

#	INVARIANT	ENFORCEMENT
INV-1	The Router holds no funds between transactions; its balance at rest is zero.	Baseline-scoped sweeps and payout (§11); `invariant_RouterHoldsNothing`, `HoldsNothingBeyondTheSeed`
INV-2	Every swap is atomic: no state exists in which part of a trade settled.	EVM transaction semantics; one frame
INV-3	No oracle: no external price feed is an input to admission, ordering, splitting, flooring or fees.	No feed call exists in the five contracts

#	INVARIANT	ENFORCEMENT
INV-4	The fee rate, the fee split and the floor constants have no setters in the deployed bytecode.	Compile-time constants in the Core and the Router (§13)
INV-5	Caller-supplied route fields can tighten protection, never loosen it.	E16 three-way maximum; the coverage gate lifts an attestation, never lowers it (§11)
INV-6	A swap with a zero minimum does not execute.	`RouterE(10)` at every door
INV-7	Ψ ranks and truncates candidates; no quote, floor or allocation reads it.	Data flow: nothing downstream of ranking consumes Ψ (§9.6)
INV-8	A pair's registry holds at most sixteen pools, and eviction keeps the fittest by full Ψ with the margin of E13.	Hub `_canInsert`; `invariant_neverExceedsMaxSlots` (the cap); `test_RecordSwap_EvictsWeakestWhenFullAndNewcomerClearsMargin` (the eviction)
INV-9	No address without runtime bytecode enters a route.	`hasCode` on every derivation (§7)
INV-10	The universal callback pays only the pool committed for the current leg, only while a leg is in flight, and never more than the leg's recorded budget.	Transient leg context; `RouterE(6)` for the pool, `RouterE(8)` for the budget (§11.4)
INV-11	A V4 hook whose address declares delta-altering permissions is rejected before any token moves, without calling it.	E18; `RouterE(9)`; Solver `_topKPools`
INV-12	No pricing product can overflow: every dangerous multiplication is factored through 512-bit `mulDiv`, and a state that cannot be bounded quotes zero.	Core (§4.2, §6)
INV-13	Rounding runs against over-promise: quotes and payouts floor, protective floors and the fee round up.	Core `mulDiv` / `mulDivUp` at each site
INV-14	Where a bound cannot be proven, the computation returns zero and the venue is surrendered — never a guess, never a schedulable revert inside a quote.	Fail-closed posture across the Core; `_solKFits` (§6)
INV-15	The fee base is the measured pull, capped by the route's committed input — never a declared output; `route.totalOut` is read by nothing.	`_chargeHopFee`; `test_FeeBase_IgnoresLiedAboutTotalOut`
INV-16	Renunciation is one-way: the flag is written `true` at one site and `false` at none; after it, only the grow-only curator tier remains, and a moved factory or hook cannot be re-armed.	`renounceControl` on Hub and Router (§14.7)
INV-17	A declared constant-product or Solidly fallback fee is believed only within a 100 bps ceiling; outside it the canonical 30 bps prices the pool.	E20

#	INVARIANT	ENFORCEMENT
INV-18	No hop carries more than four legs and no route more than eleven in the Solver's plan; the Router accepts at most three hops and five legs per hop.	Solver budgets; Router `_execute` → `RouterE(3)`
INV-19	A pool enters the registry only if it proves it trades the pair: `token0()` / `token1()` for pair-shaped kinds, re-derivation of the pool id for V4; an asked pool proves the same before discovery lists it.	Hub `recordSwap`, `_probe`; `invariant_ActiveEntriesTradeThePair`
INV-20	A settlement pays the protocol fee exactly once on an anchored route and once per hop on an exhausted one; a settlement that paid nothing does not exist.	Fee ledger, `RouterE(15)` / `RouterE(16)`; `invariant_SettledSwapEmitsExactlyOneFee` (anchored); `test/FeeSeals.t.sol`, `test/ExhaustionRegimePreviewParity.t.sol` (exhaustion)
INV-21	Hops chain — each hop's input is the previous hop's output — and no hookless leg executes after a hooked one, in this or any later hop (`sawHooked` is route-wide).	Router `_execute` → `RouterE(3)`; `sawHooked`
INV-22	The four deployed contracts — Hub, Solver, Router, Quoter — ship under the project's 24,000-byte gate on the release profile, the Core library is measured separately (6,477 bytes of 24,576), and the suite runs on that profile.	`DeployedSizeGate`; `profile_parity.py`

Appendix C · Reproduction commands

Every number in this paper that comes from the repository is reproduced by one of the following, from a clean checkout of `Blaze-Phoenix-Dex` at `8949a9d` with Foundry and Python 3 installed; fork suites need an archive RPC key in `DRPC_KEY`.

```
forge test --no-match-path 'test/fork/**'                        # the suite, release
settings (§15.1)
forge test --match-path 'test/fork/**'                           # the fork lane, five
chains (§15.1, §17)
FOUNDRY_PROFILE=release forge build --sizes --skip test --skip script   # sizes against
EIP-170 (§16.3)
python3 .github/scripts/mutants.py                                # the curated mutation
guard, 268/268 (§15.2)
python3 .github/scripts/check_targets.py                          # every mutant still
points at one line
python3 .github/scripts/assurance/covering_array.py --check       # both covering arrays
current (§15.4)
forge test --match-contract RegimeCoverageTest -vv                # strength 2, 63 rows
forge test --match-contract RegimeCoverageT3Test -vv              # strength 3, 168 rows
forge test --match-path test/regime/HostileVenueMatrix.t.sol      # ten pathologies × two
doors (§15.5)
forge test --match-path test/regime/SandwichCurve.t.sol -vv       # the sandwich curve
(§14.3)
forge test --match-path test/regime/CanonicalOracles.t.sol        # the quote maths
against the specifications (§15.6)
forge test --match-path 'test/*MetamorphicRelations.t.sol'        # 14 + 5 relations
(§15.7)
FOUNDRY_PROFILE=nver1 forge build --skip '*.t.sol' && FOUNDRY_PROFILE=nver2 forge build
--skip '*.t.sol'
NVERSION_LANE=1 forge test --match-path 'test/nversion/*'         # the three binaries
agree (§15.8)
forge test --match-test invariant_ --fuzz-seed 0xb1a2ef00          # the stateful cam-
paigns (§15.3)
forge test --match-path test/FeeSeals.t.sol                       # the fee rule, both
regimes (§13.4)
forge test --match-path test/QuoteDelayStatistics.t.sol -vv       # how a quote ages, 240
samples (§12.2)
forge test --match-path test/QuoterGasStatistics.t.sol -vv        # what a quote costs
(§12.3)
forge test --match-path test/fork/QuoteDelayFork.t.sol -vv        # live Base, 1,000 USDC
→ WETH (§12.2)
forge test --match-path test/fork/FidelityMatrix.t.sol -vv        # preview → delivery,
four chains (§17.2)
forge test --match-path test/fork/DiscoveryColdWarm.t.sol -vv     # cold and warm solve
(§17.4)
forge test --match-path test/EquationBench.t.sol -vv              # ρ* (§17.6)
forge test --match-path 'test/fork/Deployed*.t.sol'               # what V1 is, pinned by
codehash (§19)
python3 .github/scripts/assurance/projection_distance.py          # 86 / 99 (§15.9)
python3 .github/scripts/assurance/metrics.py                      # every assurance num-
ber, with its denominator
bash .github/scripts/shared-quantities.sh                         # the register agrees
with the tree
```

The hashes of Worked example 4 are reproduced by any keccak-256 implementation over `abi.encode(currency0, currency1, fee, tickSpacing, hooks)`; the paper's figures were produced with a pure-Python Keccak-f[1600] checked against the empty-string and `"abc"` test vectors and against the `extsload(bytes32)` selector the Core hard-codes.

Glossary

The operators.

SYMBOL	NAME	ONE LINE
Ω	Eightfold Dispatcher	prices every leg — closed form where a closed form is exact, ask-the-venue where the venue's own bytecode is the only honest source; "Eightfold" by history, six live kinds today
$\mathcal{D}$	Deterministic Derivation	computes pool addresses by CREATE2 arithmetic where deployment is reproducible, asks and then proves the pair where it is not, derives and proves pool ids for the V4 singleton; `hasCode` discards what has no bytecode
Φ	Iron-Law floor	the protocol's own retention floor: 96 % base, loosening with measured impact and measured concentration, hard-clamped at 80 %, applied to the final hop's in-frame quote
Ψ	Vitality Field	venue quality computed from the Monoslot alone; ranks and truncates the candidate set — it can hide a venue, never misprice one
Ξ	anti-scam projector	removes any route touching a delta-altering or inadmissible hook from the maximisation, before ranking, without calling the hook

The terms. Anchored regime — the fee charged once, on the first bridge coin the Router holds (or on the output of a direct route into a bridge coin). **Believability band** — ±5 % around the depth-weighted median of the candidates' marginal rates; inert on a candidate set of one. **Coverage gate** — a leg's attested bound lifted to the measured quote when the attestation covers less than half of it. **Exact Pass** — the Quoter's pool-exact preview: the venue's own swap run and rolled back by revert-extraction; net of the fee since 2026-09-03. **Exhaustion regime** — no bridge coin as any hop's input: every hop pays 28 bps on its own measured input; immune to a value-less prefix by construction. **Fee ledger** — a transient count of fee payments per settlement, refusing zero and refusing two on an anchored route. **Metrological Design** — build the contract as an instrument: collapse diversity into coordinates, measure every coordinate that can be measured, make the rest monotone. **Meta-Equation** — the whole aggregator as one maximisation, a product so that a single failing factor zeroes the route. **Monoslot** — a pool's entire routing state in one 256-bit word, no price inside. **One-Callback Doctrine** — one `fallback` answers every V3-shaped venue's flash-accounting callback, guarded three ways. **One-Way Door** — renunciation as bytecode: the control tier dies, the grow-only curator tier remains. **Pair-proof** — a pool proves its `token0` / `token1` (or its re-derived id) before the registry stores it or discovery lists it. **Projection distance** — how far a refusal's observed object sits from the object that decides. **Self-Healing Registry** — the protocol learns liquidity by trading it: sixteen slots per pair, fitness-ranked eviction, decay by arithmetic. **Surplus Rule** — the protocol's take is 28 bps of one measured base and nothing that scales with the gap between quote and delivery. **Third way** — any outcome of a fixture that is neither a settlement inside the floors with nothing left on the Router nor a refusal with a selector of ours.

References

1. G. Angeris, A. Evans, T. Chitra, S. Boyd. *Optimal Routing for Constant Function Market Makers.* arXiv: 2204.05238, 2022 (ACM EC '22). — Routing over CFMMs as a convex program; the optimum the proportional allocator of §10.4 sits short of.

2. T. Diamandis, M. Resnick, T. Chitra, G. Angeris. *An Efficient Algorithm for Optimal Routing Through Constant Function Market Makers.* arXiv:2302.04938, 2023 (Financial Cryptography 2023). — The dual-decomposition algorithm for the same problem (§23).
3. H. Adams, N. Zinsmeister, D. Robinson. *Uniswap v2 Core.* 2020. https://uniswap.org/whitepaper.pdf — The constant-product reference and the 30 bps default of §4.2.
4. H. Adams, N. Zinsmeister, M. Salem, R. Keefer, D. Robinson. *Uniswap v3 Core.* 2021. https://uniswap.org/whitepaper-v3.pdf — Concentrated liquidity, `slot0`, Q64.96 fixed point (§4.2, §15.6).
5. Uniswap Labs (H. Adams et al.). *Uniswap v4 Core.* 2024. https://uniswap.org/whitepaper-v4.pdf — The singleton PoolManager, flash accounting and the hook-permission address encoding (§5, §14.2).
6. M. Egorov. *StableSwap — Efficient Mechanism for Stablecoin Liquidity.* 2019. https://curve.fi/files/stableswap-paper.pdf — The amplified invariant of the family excised in §4.1.
7. P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, A. Juels. *Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability.* IEEE S&P 2020. DOI 10.1109/SP40000.2020.00040. — The ordering adversary §14.3 bounds rather than claims to eliminate.
8. L. Zhou, K. Qin, C. F. Torres, D. V. Le, A. Gervais. *High-Frequency Trading on Decentralized On-Chain Exchanges.* IEEE S&P 2021. DOI 10.1109/SP40001.2021.00027. — Sandwich attacks measured and formalised (§14.3).
9. P. Cousot, R. Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* POPL 1977. DOI 10.1145/512950.512973. — The frame for the single-tick closed form as a sound abstraction (§23).
10. M. R. Clarkson, F. B. Schneider. *Hyperproperties.* Journal of Computer Security 18(6), 2010. DOI 10.3233/JCS-2009-0393. — Properties of sets of traces; the class the redistribution findings belong to (§21, §23).
11. T. Y. Chen, S. C. Cheung, S. M. Yiu. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* HKUST-CS98-01, 1998; arXiv:2002.12543. — The second judge of §15.7.
12. R. A. DeMillo, R. J. Lipton, F. G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer.* IEEE Computer 11(4), 1978. DOI 10.1109/C-M.1978.218136. — Mutation analysis and the coupling hypothesis (§15.2).
13. A. Avizienis. *The N-Version Approach to Fault-Tolerant Software.* IEEE TSE SE-11(12), 1985. DOI 10.1109/TSE.1985.231893. — The ancestor of §15.8.
14. D. R. Kuhn, R. N. Kacker, Y. Lei. *Practical Combinatorial Testing.* NIST SP 800-142, 2010. DOI 10.6028/NIST.SP.800-142. — Covering arrays and t-way interaction coverage (§15.4).
15. B. Littlewood, L. Strigini. *Validation of Ultrahigh Dependability for Software-Based Systems.* Communications of the ACM 36(11), 1993. DOI 10.1145/163359.163373. — Why no probability of correctness is stated (§15.11).
16. A. Akhunov, M. Salem. *EIP-1153: Transient Storage Opcodes.* https://eips.ethereum.org/EIPS/eip-1153 — The leg context, the fee ledger and the reentrancy lock (§11.4, §13.2).
17. V. Buterin. *EIP-170: Contract Code Size Limit.* https://eips.ethereum.org/EIPS/eip-170 — The 24,576-byte wall (§16.3).
18. V. Buterin. *EIP-1014: Skinny CREATE2.* https://eips.ethereum.org/EIPS/eip-1014 — The address arithmetic of $\mathcal{D}$ (§7).

19. M. Lundfall et al. *EIP-2612: Permit Extension for EIP-20 Signed Approvals.* https://eips.ethereum.org/EIPS/eip-2612 · Uniswap Labs. *Permit2.* https://github.com/Uniswap/permit2 — The Permit2 door (§11.1).
20. V. Buterin, S. Wilson, A. Dietrichs, M. Garnett. *EIP-7702: Set Code for EOAs.* https://eips.ethereum.org/EIPS/eip-7702 — Delegated accounts call the same doors (§11.1).
21. *ERC-7201: Namespaced Storage Layout.* https://eips.ethereum.org/EIPS/eip-7201 — The Hub's storage namespace (§16.1).
22. MariaDB Corporation Ab. *Business Source License 1.1.* https://mariadb.com/bsl11/ — The code licence.
23. Fable & Mitra. *BlazePhoenix-Dex: an on-chain DEX aggregator with measured routing.* Repository, `main` 8949a9d, 2026-09-05. https://github.com/blazephoenixxyz-crypto/Blaze-Phoenix-Dex — The canonical source for every claim in this paper; Appendix C reproduces the numbers from it.
24. S. Müller, N. Moumeni, A. Messaoudi. *Cost of Manipulation in AMM-Based Oracles.* arXiv:2606.03548, 2026 (Financial Cryptography 2026, DeFi workshop). — The liquidity-weighted median bound behind E25 (§10.3, §23).
25. *Ormer: A Manipulation-resistant and Gas-efficient Blockchain Pricing Oracle for DeFi.* arXiv:2410.07893. — A median-estimating on-chain oracle (§23).
26. *From Means to Medians: Optimal Benchmark Design.* arXiv:2506.22142. — Which benchmarks are hardest to manipulate (§23).
27. *Optimal Routing across Constant Function Market Makers with Gas Fees.* arXiv:2603.02844. — Routing with gas (§23).
28. *Routing Frictions and Executable Liquidity in Fragmented Markets.* arXiv:2609.19013. — Advertised versus executable liquidity (§23).