

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 2.4.0

Written by ***Mitra***

Version	2.4.0 · supersedes 2.3.1
Date	2026-11-23
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. 7

Abstract 7

How to read this paper 8

The Innovations 8

I-1 · Same-state quote equivalence 9

I-2 · The on-chain solver, free to call 9

I-3 · The Meta-Equation 9

I-4 · The Eightfold Dispatcher Ω 9

I-5 · Deterministic Derivation $\mathcal{D}$ 10

I-6 · Uniswap V4, entirely in-frame 10

I-7 · The Solidly stable solver 10

I-8 · The Monoslot 10

I-9 · The Self-Healing Registry and the Vitality Field Ψ 11

I-10 · The Capital-Anchored Split 11

I-11 · The Iron-Law floor Φ 11

I-12 · Measurement at every seam 12

I-13 · The One-Callback Doctrine 12

I-14 · The Surplus Rule and immunity by exhaustion 12

I-15 · The byte–gas exchange rate ρ^* 12

I-16 · The anti-scam projector Ξ 13

I-17 · The One-Way Door 13

I-18 · Metrological Design and the published denominator 13

I-19 · The Gate: a venue is data behind an immutable wrapper [candidate: first must be falsifiable] 13

I-20 · Monotonicity against the caller [candidate: first must be falsifiable] 14

I-21 · One question, one producer [candidate: first must be falsifiable] 14

The scoreboard 14

Part I — The problem and the discipline 15

§1 The problem: a quote is a promise someone else computed 15

§1.1 The fragmentation is real, and shallower than it looks 16

§1.2 The off-chain escape hatch 16

§2 The answer and the discipline 17

§2.1 Invariant-driven design 18

§2.2 Why this discipline fits this medium 19

§2.3 Metrological Design: one morphism, measured coordinates 19

§2.4 Measure, re-home, or make lying costly 21

§2.5 One question, one producer 21

§2.6 Membership is not acceptance 22

§2.7 A guard that cannot fire 22

§2.8 Fix the invariant, not the instance 23

§2.9 Verifying beats solving; measuring beats recomputing 23

§3 The Meta-Equation and the dependency budget 23

§3.1 The dependency budget 25

§3.2 Three edges ... 26
Part II — The primitives ... 26
§4 The Eightfold Dispatcher Ω ... 26
§4.1 Seven kinds, five families, one attribute table ... 26
§4.2 The closed forms ... 28
§4.3 Two fee units, carried natively ... 30
§4.4 One dispatcher, three readers ... 30
§4.5 Venues behind the Gate ... 31
§5 Uniswap V4 end to end ... 32
§5.1 Locating a pool that has no address: derive, then prove ... 32
§5.2 Reading a pool that answers no calls ... 35
§5.3 Settling with a singleton ... 35
§5.4 The fee that exists only at execution time ... 36
§5.5 Hooks: screened by arithmetic, never by interrogation ... 37
§5.6 What the integration is worth ... 37
§6 The Solidly solver ... 38
§6.1 Primary path: ask the pool ... 39
§6.2 Fallback: replicate, then under-ask ... 39
§6.3 The solve ... 40
§6.4 Decimal normalisation ... 41
§6.5 Fail-closed on the whole domain ... 41
§6.6 The measured residual, and the 200 bps margin ... 42
§7 Deterministic Derivation $\mathcal{D}$ ... 43
§7.1 The mechanism and the table ... 43
§7.2 A theorem, a claim, and a proof ... 44
§7.3 Two subtleties in the V3-shaped rows, and one in the asked ones ... 45
§7.4 The final gate, and where deriving stops ... 46
§7.5 Discovery consumes the table row by row ... 46
§8 The Monoslot ... 47
§9 The Self-Healing Registry and the Vitality Field Ψ ... 49
§9.1 Learning by trading, and the pair-proof ... 50
§9.2 The score ... 51
§9.3 Eviction ... 53
§9.4 Discovery, and its gas knob ... 53
§9.5 The configuration surface ... 53
§9.6 The structural boundary ... 54
§26 The machine underneath ... 54
§26.1 The size gate, measured in the build ... 54
§26.2 The transient lock ... 55
§26.3 The callback is authenticated by a committed pool ... 56
§26.4 Assembly where it is measured ... 57
§26.5 A stateless library, pinned by hash ... 57
Part III — Routing and execution ... 58

§10 Routing and the Split Allocator ... 58
§10.1 Route shapes and budgets ... 59
§10.2 The funnel ... 59
§10.3 The believability band ... 60
§10.4 The allocator and its four guards ... 61
§11 The execution layer and the Iron-Law floor Φ ... 63
§11.1 Four doors, one executor ... 63
§11.2 Measurement, from the first instruction ... 64
§11.3 Four refusals under every leg ... 65
§11.4 One callback, one unlock ... 65
§11.5 The Iron-Law floor Φ, in full ... 66
§11.6 Settlement, and how to read one ... 68
§11.7 The Router's refusals ... 69
§11.8 Fee-on-transfer on the way out ... 70
§12 Quote surfaces, and how a quote ages ... 71
§12.1 The Quoter's surfaces ... 71
§12.2 How a quote ages ... 74
§12.3 What a quote costs through the ABI ... 76
§12.4 The in-frame budgets ... 76
§13 Fee policy: the two regimes and the Surplus Rule ... 76
§13.1 Where the fee is charged: two regimes, both named ... 77
§13.2 The commitment has one producer, and the contract counts ... 78
§13.3 The Surplus Rule ... 78
§13.4 The fee, measured from outside the Router ... 79
Part IV — Security and verification ... 82
§14 Security model ... 82
§14.1 Attacks and the checks that refuse them ... 82
§14.2 Hostile hooks: three layers, cheapest first ... 86
§14.3 Bounded, not eliminated: staleness, ordering, and the sandwich curve ... 88
§14.4 Hostile venues ... 89
§14.5 The hostile front end ... 90
§14.6 What a hostile key can and cannot do ... 90
§14.7 The One-Way Door ... 91
§14.8 What this model asks of you ... 92
§14.9 The guarantees, each with its own check ... 92
§15 The verification apparatus, measured ... 93
§15.1 The suite in numbers ... 93
§15.2 The evidence chain, and the curated mutation guard ... 95
§15.3 Mutants aimed at the invariants ... 95
§15.4 Regime covering arrays ... 96
§15.5 The hostile-venue matrix ... 97
§15.6 Canonical oracles ... 98
§15.7 Metamorphic relations ... 99
§15.8 N-version: the same source under other code generation ... 100

§15.9 Projection distance ... 101
§15.10 Evidence independence, the shared-quantity register, and the artefact ... 102
§15.11 What none of this establishes ... 102
Part V — Architecture, performance and deployment ... 103
§16 Architecture and sizes ... 103
§16.1 The contracts: what each holds, what each may call ... 104
§16.2 The life of one swap ... 106
§16.3 Sizes against EIP-170 ... 107
§16.4 The attack surface, counted ... 108
§17 Measured performance ... 110
§17.1 Cross-chain identity ... 111
§17.2 Preview to delivery fidelity ... 111
§17.3 The factory census and the public sweep ... 112
§17.4 Gas ... 113
§17.5 The executor against the field ... 115
§17.6 The byte–gas exchange rate ρ^* ... 115
§18 The DEX, stripped to the bone ... 116
§19 Deployment status ... 118
§27 Interfaces: the SDK, the MCP servers, the API and the Hermes plugin ... 119
§27.1 The common model ... 119
§27.2 The SDK ... 119
§27.3 The local MCP server ... 121
§27.4 The hosted MCP and the API ... 122
§27.5 The Hermes plugin ... 122
§27.6 The APP ... 123
§27.7 Summary, and what was not checked ... 124
Part VI — Staking V4 ... 125
§25 Staking V4: Design ... 125
§25.1 Architecture and roles ... 126
§25.2 The flow: stake, unstake, claim, borrow ... 127
§25.3 Reward accounting ... 129
§25.4 Interest: the curve and the index ... 133
§25.5 The reserve, solvency and the loss waterfall ... 134
§25.6 Hardening in V4 ... 136
§25.7 Test evidence ... 136
§25.8 Two deployments, one schedule each ... 137
§25.9 Clocks: seconds, blocks and chains ... 138
§25.10 Reading solvency yourself ... 139
§25.11 The circuit breaker ... 141
Part VII — The token, the research programme and what you must trust ... 143
§20 Token — BZPX ... 143
§21 Research programme ... 144
§22 What you must trust, and what you need not ... 145

§23 Related work 147
§24 Conclusion 148
Revision history 148
Version 2.4.0 (2026-11) 148
Version 2.3.0 (2026-10-14) 149
Version 2.3 (2026-09-24) 149
Errata to 2.2 (2026-09-07) 150
Errata to 2.2 (2026-09-24) 151
Appendix A · The primitive equations 151
Appendix B · The invariants 155
B.1 The numbered invariants 155
B.2 The catalogue of stateful invariants 158
B.3 Symbolic proofs (Halmos) 164
B.4 Properties of the compiled artefact 166
B.5 Specified properties 166
Appendix C · Reproduction commands 169
Appendix D · The shared-quantity catalogue 171
Appendix E · The venues 182
Glossary 183
References 185
How to cite 187

> *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 user balance at rest.** Between trades the Router holds no user funds; the only balance it keeps is a fee share a token refused to deliver, booked to its treasury slot.
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: quote and execution run the same on-chain computation, so at the same chain state they give the same output, and if the state moves the swap delivers at least the reviewed minimum or reverts. 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 dispatcher in the executing frame. There is no routing service, no price oracle, no keeper and no upgrade path. This paper specifies the seven contracts — Router, Solver, Hub, Quoter, Core, Gate and Math — 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, which this edition sets out as a chapter of its own (§2): measure every coordinate that can be measured, re-home what cannot be measured to a source no caller controls, and make the rest monotone so that a false input hurts only its supplier. A venue outside the native families enters through the Gate, behind an immutable wrapper with a 48-hour arming delay (§4.5). Every guarantee is stated with its enforcement site and its limit. The verification apparatus is printed with its denominators: at DEX commit `969bcc6`, 54 stateful invariants across 18 files, 1,895 registered mutants and 4 Certora specifications, with 3,517 declared test functions in 515 files (§15.1); 15 Halmos checks report `[PASS]` in the CI run at commit `6387350` (Appendix B.3). Eighteen named innovations carry the design, from the Meta-Equation to immunity by ex-

haustion; each is catalogued with its mathematics before the specification begins. The staking engine of Part VI is specified as two independent deployments, each with its own contract, reserve and schedule. The generation live on Base, Ethereum, Optimism, Arbitrum and Robinhood Chain is V1; this paper specifies V2, a pre-launch engineering preview with no mainnet deployment yet. V2 targets six networks (Base, Optimism, Arbitrum, Robinhood Chain, Ethereum L1 and Avalanche) and launches 15 December 2026. At the commit this paper cites, the fork lane runs its fidelity matrix on four chains (Base, Optimism, Arbitrum and Robinhood Chain): it compares each executed swap with its quote on forks of live liquidity, and every comparison it prints is 0 bps (§17).

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. The paper has seven Parts: I, the problem and the discipline (§1–§3); II, the primitives (§4–§9 and §26); III, routing and execution (§10–§13); IV, security and verification (§14–§15); V, architecture, performance and deployment (§16–§19 and §27); VI, staking V4 (§25); VII, the token, the research programme and what you must trust (§20–§24). Three appendices are catalogues you can read on their own: Appendix B (invariants: statement, where it applies, what it measures, the test that checks it, the mutant that breaks it), Appendix D (the shared primitives, each with the one question it answers and its single producer) and Appendix E (venues: kind, how found, how measured, closed form or delegated). 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. Numbering is an identifier, not an order: the section, innovation, equation and invariant numbers of Version 2.3 are frozen, because articles, the machine-readable facts and other papers cite them. What is new takes a new number — a subsection (§2.4–§2.9, §4.5, §11.8, §25.8–§25.10) or a chapter (§26, §27) in the Part where it belongs — and a number is never reused. Two repositories are in scope, the DEX and the staking contract, each cited at its commit; 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. Three candidates, I-19 to I-21, follow the eighteen: each is marked until its claim of a first has been shown falsifiable.

I-1 · Same-state quote equivalence

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 at `969bcc6` (2026-11-16): in a 240-sample delay study, all 64 samples whose state did not move between quote and settlement delivered the quoted output (**64 / 64**); every other sample delivered at least its attested floor or reverted. (§3, §11, §12)

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

What. `Solver.findBestRoutePlan(tokenIn, tokenOut, amountIn)` is a `view` function. Anyone gets the route the public code finds 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. Routing is usually computed on a server you cannot inspect. Ours is 31,933 bytes of public bytecode (Solver 21,253 and Math 10,680 at commit `969bcc6`) 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.

$$R^\star = \arg\max_{R} \ \mathrm{net}(R) \cdot \mathbf{1}[\mathrm{net}(R) \geq \Phi(R)] \cdot \Xi(R)$$

Why it matters. The product form is a safety property: a single failing factor — a leg quoting zero, a 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 native venue kinds — Uniswap V2, V3, V4 (plain and native-currency), Algebra and Solidly — collapse into four mathematical families behind one attribute table and one dispatcher; a seventh kind, external, prices a venue whose own bytecode is the only honest source, behind an immutable wrapper at the Gate (§4.5).

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 reviewed 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. $\text{pool} = \text{addr}_{20}(\text{keccak256}(\texttt{0xff} \,\|\, \text{origin} \,\|\, \text{salt} \,\|\, \text{initCodeHash}))$ (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 $= \text{keccak256}(\text{abi.encode}(\text{poolId}, 6))$ 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 $k(x, y) = x^3y + xy^3$ (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 asks the pool when it can, and otherwise replicates the curve and under-asks by a measured margin — 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, own tick rhythm — packed into **one 256-bit word**.

Math. $s = a \mid f \ll 8 \mid k \ll 32 \mid r \ll 40 \mid (c \wedge \texttt{0xFFF}) \ll 48 \mid b \ll 60 \mid \ldots \mid \tau \ll 224$ (E9), the top field being the pool's own tick rhythm τ (it held the last block in the 2.3).

Why it matters. One `SLOAD` reads a pool's ranking state (the bridge bonus is read live); 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 $h = 4\tau$ seconds of silence, where τ is the pool's own moving mean of the time between its anchored ticks (the old fixed 24,576 s step survives only while τ is unmeasured), $V = \max(1, \lfloor c \cdot 24{,}576/h \rfloor)$ for a live slot (a dead one scores 0), with $c = \text{swapCount} \gg \lfloor (t - t_{\text{last}})/h \rfloor$ less a chord term inside the step, and the pool is dead past 31 halvings (E11, §9.2); depth class $b = \min(15, \lfloor \log_{10}(d/10^{15}) \rfloor)$ (E12); a newcomer displaces the weakest only past a margin, $\Psi_{\text{new}} = 2^{b(d_{\text{new}})}\,(1 + 0.25\,\mathbf{1}_{\text{bridge}}) > \Psi_{\text{worst}} + \lfloor \Psi_{\text{worst}}/4 \rfloor$ (E13, §9.3).

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 a band whose width is set by no-arbitrage from each venue's fee and measured error, not by a number of basis points (E25, §10.3); 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 hold more than half of the depth that votes on the pair, more than all its honest pools together, measured from reserves and liquidity that a donation does not move (`_depthWeightedMedian`; breakdown point 50 %) — 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 at commit `969bcc6` on four chains, the Solver's route is never more than 0.01 bps below the grid's best (Arbitrum −0.0072 bps; Optimism and Robinhood equal to the wei; on Base the Solver beat the grid by 1.18 bps). (§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. When an attacker moves 1 % of the pool's depth around the trade, the victim loses 2.22 % against the quote; from a 3 % move the floor **refuses** and the attacker's round trip turns negative (CI at `969bcc6`). (§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 **no user funds at rest** between transactions (INV-1); the only balance it keeps is a fee share a token refused to deliver, booked to its treasury slot. (§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 amount is read from the signs of the two deltas, the token from the in-flight leg context; 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. Where the fee is charged on an input, everything the pools deliver above the quote reaches you in full; on a direct route into a bridge coin it is 28 bps of the gross delivery, surplus included. 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, and each treasury can still move its own fee slot (`moveTreasury`) and claim what that slot is owed.

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, except that each treasury can redirect its own share of the fee. (§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, re-home what cannot be measured to a source no caller controls, make the rest monotone so a false input hurts only its supplier — and a verification apparatus that reports **every number beside its denominator**.

Math. At DEX commit `969bcc6`: 54 stateful invariants across 18 files, 1,895 registered mutants and 4 Certora specifications; 3,517 declared test functions in 515 files, of which 3,562 tests ran in CI (suite and fork lane); in the CI run at that commit, 1,753 mutants ran and all were caught, each registered mutant paired with the test that must fail; covering arrays over the regime factors of strength 2 (60 rows, 237 pairs) and 3 (145 rows, 1,441 triples), every row run passing; 15 Halmos checks reporting `[PASS]` in the CI run at commit `6387350` (Appendix B.3), and properties specified in `certora/specs/` for the Certora Prover (Appendix B.5).

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, §15)

I-19 · The Gate: a venue is data behind an immutable wrapper [candidate: first must be falsifiable]

What. A venue outside the native families enters through the Gate as an immutable wrapper, one per venue, armed after a 48-hour delay, at most four wrappers per token pair. No permissionless input chooses a venue's shape: a new venue is data behind a wrapper, not a new branch in the dispatcher.

Math. `ADAPTER_DELAY` = 48 hours (`BlazePhoenixGate.sol:76`); `GATE_MAX_WRAPPERS` = 4 (`BlazePhoenixCore.sol:192`); the external kind is kind 9 (`BlazePhoenixCore.sol:320`), with a quote-gas constant `GATE_QUOTE_GAS` = 500,000 (`BlazePhoenixCore.sol:3271`), at commit `6387350`.

Falsifiable claim. No permissionless input chooses a venue's shape; a new venue family is code behind immutable wrappers, and its adapter is armed only 48 hours after it is proposed, with the code pinned at proposal (`ADAPTER_DELAY` ; `acceptAdapter` refuses before the delay or if the code moved). Every wrapper call re-reads the armed adapter and refuses when there is none or its code left the pin. It fails if any call path lets an unarmed or caller-supplied wrapper be priced.

Why it matters. The set of venues grows without the set of trusted code paths growing with it. (§4.5)

I-20 · Monotonicity against the caller [candidate: first must be falsifiable]

What. A number the caller supplies is accepted only where the function is designed so that a false value moves the result against the caller. This is the third legal exit for a caller-supplied number, beside measuring it and re-homing it to an immutable or sealed source.

Math. For a caller number c with measured value c_0, the result $u(c)$ to the caller satisfies $u(c) \leq u(c_0)$ for every c on the lying side of c_0; then no lie can beat the truth, and a monotone number may be computed off-chain without moving trust (§2.4).

Falsifiable claim. For every caller-supplied number the protocol accepts, there is a test in which the lying side yields a result no better than the measured one. It fails at the first input where lying pays.

Why it matters. Off-chain computation and honest hints cost nothing in trust, because a wrong hint costs only its sender. (§2.4)

I-21 · One question, one producer [candidate: first must be falsifiable]

What. A quantity that more than one place needs has one producer; each consumer writes down the question it asks, and where two questions differ without a parity pin, that is where the next instance of a defect would be. The register is a published instrument, catalogued in full in Appendix D.

Math. At the pinned commits, the DEX shared-quantity register has 131 lines in 5 sections and the staking register has 37 lines (`SHARED_QUANTITIES.md` , `969bcc6` and `9958d5f`).

Falsifiable claim. Every quantity computed in two places appears in the register with the question each consumer asks and a named test that fails when the answers diverge. It fails at a second computation the register does not list.

Why it matters. The seam between two correct computations of one number is where a measured-versus-modelled drift hides; naming the producer closes it. (§2.5)

The scoreboard

Rows marked 2.3 carry commit `8916545` (measured 2026-09, Version 2.3); the others are read from the repositories or the CI run at `969bcc6` .

MEASURED PROPERTY	RESULT
Stateful invariants in the DEX test tree (`969bcc6`)	**54 in 18 files**

MEASURED PROPERTY	RESULT
Mutants in the DEX mutation harness (`969bcc6`)	**1,895** registered; 1,753 run in CI at `969bcc6` , all caught
Halmos checks reporting `[PASS]` (CI run at `6387350` , Appendix B.3) · Certora specifications (`969bcc6` ; Certora: specified, Appendix B.5)	**15 · 4**
Samples whose state did not move between quote and settlement that delivered the quoted output (240-sample delay study)	**64 / 64**
Venue pathologies × doors that settle, refuse with a selector of ours, or revert whole with the balance untouched (`969bcc6`)	**20 / 20**
Control actions whose refusal is exercised with its own revert code (2.3)	**31 / 31**
Quantities computed in two places that a named test ties together (`969bcc6`)	**11 / 11**
Regime-factor pairs in the strength-2 covering array; rows run, all passing (`969bcc6`)	**237 pairs · 60 / 60**
Regime-factor triples in the strength-3 covering array; rows run, all passing (`969bcc6`)	**1,441 triples · 145 / 145**
Optimiser builds of the quote maths in agreement, runs 1 / 300 / 20,000 (2.3)	**3 / 3**
Preview → delivery across four chains (`969bcc6` , 12 rows, one Base row not executed)	**0 bps**
Solver route vs. an exhaustive grid, four chains (`969bcc6`)	**never more than 0.01 bps below the grid's best**
Victim's loss against the quote when an attacker moves 1 % of depth (`969bcc6`); the floor refuses from a 3 % move	**2.22 %**

Part I — The problem and the discipline

§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 (survey not re-checked for this edition) — 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 no user balance at rest (only a fee share a token refused to deliver, owed to its treasury slot); 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.

One law sits under the four decisions and under everything in this chapter: no permissionless datum may choose a venue's shape or domain, the `(factory, init-code-hash)` pair that identifies it, or lower a floor; security parameters come only from an immutable or a sealed source, and what a proposer supplies are coordinates that the contract measures on chain and fails closed on. §2.1 to §2.3 state the method that follows from it, and §2.4 to §2.9 state the six working rules it has produced, each with the guard that enforces it.

§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 user funds between transactions; the only balance it keeps is a fee share a token refused to deliver. 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:

PRIN-CIPLE	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.
Fail closed	When a property cannot be established, 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

Invariant-driven design says what must always hold. 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.*

The unifying law and the measurement principle. No permissionless datum may choose a shape, a domain or a `(factory, init-code-hash)` pair, or lower a floor; a security parameter comes only from an immutable or a sealed source, and whatever a proposer supplies is a coordinate the contract measures on chain and fails closed on. The working form of the law is the measurement principle: the contract never acts on a modelled, nominal or assumed value when an on-chain measurement exists. The principle has three corollaries. First, a measurement is sampled where the attacker does not control the sampling: a value the attacker

can set at the moment you read it is a claim, not a measurement. Second, the absence of a measurement is not a value: an empty, zero or missing read is a fact about the source, so the candidate is dropped or refused and never replaced by a default. Third, the instrument must be fixed or observable: a mutable instrument with no event measures the wrong place, so an instrument is compiled in, or its change is visible before it takes effect (§4.5).

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: 25 rows (18 distinct field names, some listed once per pool family), 17 confirmed against an observation, 5 steering, 3 declared with the reason (§15; `docs/assurance/fields.json`).

Where measurement ends. Some numbers can be neither measured nor re-derived: a caller's floor, a caller's declared scale, the caller's own minimum. §2.4 states the three legal exits for such a number and the one that applies when measuring is impossible.

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 (`MIN_SPLIT_IMPROVEMENT_PPM`), about eight times the 3 ppm that an extra leg's 320 bytes of calldata cost on Base (§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.

§2.4 Measure, re-home, or make lying costly

Some numbers enter the system from a caller and cannot be measured by the contract: the minimum a caller will accept, the scale a caller declares, the route a caller proposes. The discipline allows exactly three exits for such a number, and no fourth.

1. **Measure it.** Read the quantity from the pool, from a balance delta, or from a derivation, and discard the claim. This is the first exit and the one used wherever a measurement exists (§2.3).
2. **Re-home it.** Move the quantity to a source the caller cannot write: an immutable constant, a value sealed at construction, or a record only the protocol's own code can update. The parameters that bound security (the floors of §11, the arming delay of §4.5, the registry caps of §9) live here.
3. **Make lying costly.** Accept the caller's number and design the function that consumes it so that a false value can only hurt the party who supplied it.

The third exit is the least obvious and the one worth deriving. Let c^* be the value the caller would honestly choose, let C be the values the caller is able to pass, and let $G(c)$ be the guarantee the protocol makes to everyone else when c is passed. The function is *monotone against the caller* when

$$G(c) \;\geq\; G_0 \quad \text{for every } c \in C,$$

where G_0 is the guarantee that holds with no caller input at all, and the caller's choice can only move the outcome *toward* the caller's own loss. The settlement floor is the concrete case. The Router delivers at least $\max(F,\, m)$, where F is the protocol floor computed from measured quantities (§11) and m is the caller's `userMinOut`. Since $\max(F, m) \geq F$ for every m, no choice of m lowers the protocol floor. Since $\max(F, m)$ is non-decreasing in m, a larger m only adds a revert condition the caller imposed on its own trade, and a smaller m falls back to F. A caller who lies about m therefore moves only its own result. The repository states the two halves as invariants, `invariant_DeliveredNeverBelowTheProtocolFloor` and `invariant_DeliveredNeverBelowUserMinOut` (Router invariant suite, DEX commit 969bcc6).

The consequence is practical. A monotone input may be computed anywhere (in the interface, in a script, by an agent) without moving trust, because the contract does not rely on the computation being honest. This is why the minimum a person sets is the one bound no interface can take from them (§3.2), and why the order in which exits are tried matters: measure first, re-home second, and reach for monotonicity only when the number is the caller's by nature.

§2.5 One question, one producer

The defect class this architecture is most exposed to is not a missing check. It is two components answering one question in two frames: a planner and an executor, a preview and a delivery, a quote and a fill. Each answer can be individually right and the pair still disagree.

The rule is that a quantity with more than one producer or consumer is written in a register, with the *question* it answers, and the register says what binds the copies: either a single producer with a CI guard forbidding a second, or a named test that asserts the producers agree and states what it pins. A CI check fails the build when a row's test does not reach the quantity the row claims to tie. At the commits this paper measures

the DEX register holds 131 rows in five sections (floors and fees, registry state, identity and keys, transient state, measurement harness) and the staking register 37 rows [SHARED_QUANTITIES.md, DEX 969bcc6, STK 9958d5f]. Only totals are published here; the grading of each row is in the repository.

The agreement tests are ordinary invariants, which is what makes them checkable: `invariant_encodeAgreesWithPreview` and `invariant_accountingNeverDrifts` tie the Quoter's preview to the encoded plan, and `invariant_Sq_BothSolversAnswerTheSameBytes` ties the two solver paths to one answer.

The rule has a boundary, stated so it is not over-applied. Single production is for *primitives*: a quantity every consumer needs derived from the source. The predicates that decide acceptance keep their redundancy on purpose, because a refusal that two independent checks can each make is stronger than a refusal one check makes (§2.6).

§2.6 Membership is not acceptance

The Metrological Design collapses diversity into coordinates (§2.3). The collapse has a limit, and the limit is the difference between *membership* and *acceptance*.

Membership is a read: which pools exist for a pair, which kind each belongs to, what its state is. A membership question has one answer and one producer, so it collapses into a single table and a single derivation. *Acceptance* is a gate: whether the Router may execute across a given pool, at this size, under this floor. Acceptance stays a gate with its own reads. If the two are fused, the check that a pool is *listed* silently becomes the check that it is *safe to trade*, and a coincidence that held when the two were written becomes a dependency nobody declared. A coincidence measured today is pinned by a parity test that fails when the two producers drift, and is not merged into one.

The same honesty applies to pricing. There is no universal formula for every venue. The Core carries two closed forms, constant product and concentrated liquidity (§4.2); every other venue is DELEGATED, meaning the venue's own quote function is asked and its answer is bounded by the measured delta and the floor (§4.5). Unification reduces the if/else in the code; it does not remove the verification, and the paper does not claim that it does.

§2.7 A guard that cannot fire

A guard is a claim: *this input is refused*. A test that passes with the guard in place proves nothing about the guard unless the same test fails when the guard is removed. The rule is therefore structural: **every guard has a watcher.** The register holds, for each guard, one exact source line, the replacement that disables it, and the one test that must go red under that replacement. A guard without a mutant does not merge, and a one-second static check confirms that every mutant still points at exactly one line, so a refactor cannot silently orphan a watcher [AGENTS.md, rules 1-2; docs/AUDIT_METHOD.md §3, DEX 969bcc6].

Derivation of why the pairing is needed. Let T be a test, G a guard, and M the program with G removed. T passing on the real program shows T and the program agree. It shows that T *watches* G only if T fails on M. A suite can be large and green while no test watches a given guard; the mutant is the instrument that tells the two cases apart, which is why the register is measured and not assumed.

At commit 969bcc6 the DEX mutant register holds 1,895 mutants [README:270-276]. The result of running them is a campaign measurement and is given with its denominator in §15. The same pairing is applied to the stateful invariants themselves: a mutant is aimed at an `invariant_*` function that must go red under it, so the invariants are also watched and are not only believed (§15.3).

§2.8 Fix the invariant, not the instance

When a defect is found, the instance is not the finding. The finding is the invariant that failed to hold, and the instance is one place it failed. A patch to the instance restores that place and leaves every other place the same class can occur.

The rule is that a fix states the invariant, places the guard where *every* instance of the class passes through it, and arrives with a test over the class rather than over the one input that exposed it: a fuzz or invariant campaign, not a fixture. The test header names the commit at which it was red, so red-before-green is checkable from the history, and the guard gets its mutant (§2.7). N instances of one class therefore resolve with one fix, one test over the class and one watcher, and the next instance of the class is refused by construction.

§2.9 Verifying beats solving; measuring beats recomputing

Two laws sit under the meta-equation of §3.

Law 1: verifying beats solving. The Solver returns a plan $\hat{a} \approx a^*$, where a^* is the argmax of the meta-equation $D(x, s) = \arg\max_a [\mu(a) - \gamma\, g(a)]$. Finding a^* is an approximate search over routes and splits. The constraint the Router enforces is different in kind: the delivered amount against the floor, measured from balance deltas. That check is exact and costs O(1) in the size of the search. The safety predicate $S(\hat{a})$ is evaluated on $\hat{a}$ itself and does not mention a^*. Hence for any $\hat{a}$ that passes S, settlement is safe; and $\mu(\hat{a}) \le \mu(a^*)$ means an imperfect search costs *price*, never *safety*. This asymmetry, exact verification against approximate search, is what lets the system be safe while its routing is imperfect, and it is why the paper never claims the route is optimal: it claims the route is checked.

Law 2: measuring beats recomputing. A leg is opened only while its marginal output exceeds its marginal cost, $\partial\mu > \gamma\, \partial g$, using the marginal output measured at the point of the split. There is no fixed threshold and no magic constant to choose; the stopping rule is the inequality, evaluated on measured quantities. The one numeric gate the split carries, `MIN_SPLIT_IMPROVEMENT_PPM = 25`, is a margin priced from a measured break-even (§2.3, §10), not a tuned limit.

Section 3 states the equation, the dependency budget that follows from it, and the three edges where the claim stops.

§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:

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

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

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; $\Phi(R)$ 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	Orders the probe and the registry's eviction, from earned quality read from the pool's packed on-chain state; never a factor in the output	§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 *in what order to probe it, and whom to evict first*; 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 zeroes the route — 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.

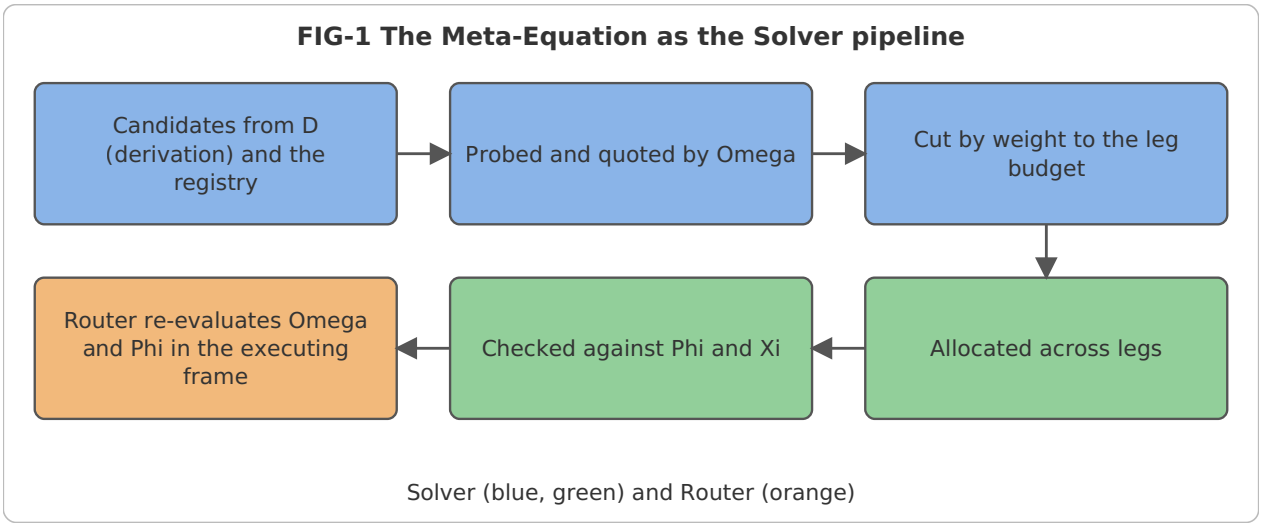

FIG. 1 The Meta-Equation as the Solver pipeline

FIG-1 — The Meta-Equation as the Solver's pipeline: candidates from $\mathcal{D}$ and the registry, probed and quoted by Ω, cut by weight to the leg budget, 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 $\mathcal{D}$; ordered by Ψ, which fixes only the order of the probe (no count cuts the list: every candidate is probed and quoted by Ω, and dust is measured against the pair, §10); filtered by a believability band; cut by weight to the leg budget, `MAX_LEGS_PER_HOP` = 5 per hop, after the probe [checked in code: S:612-618, S:888-892, C:149]; 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 the code that pipeline runs is listed in §4-§11. Ψ only orders the probe; it never decides which candidate is quoted.

§3.1 The dependency budget

Instead of adjectives — the usual ones 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 seven contracts
A solver network or auction	Common	None
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 upgrade initialiser: the Hub's one-shot `initialize` (admin-only, refused after renunciation) wires the V4 manager once; the only `delegatecall`s are the compiler's own into the public functions of the two stateless libraries, Core and Math, fixed at link time (§26.5) [checked in code: Core:3025, 3061, 3535; Hub: 1740; Math:20-23; Solver:790-795]
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 on any node, with no server of ours in the path. 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 that cannot change — 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, which refuses while paused. After renunciation the control tier is gone for ever: in the Router, admin, treasury assignment, Permit2, WETH, pause and rescue; in the Hub, pause, roles, wiring and removals; in the Gate, clearing or replacing an adapter family that was ever live. What remains is grow-only curation of venues and the payee's own slot: each treasury may move its slot and claim the fees that slot is owed (§14). Stateless core, serverless operation, grow-only governance: the table is the claim; the words are its shadow.

Part II — The primitives

§4 The Eightfold Dispatcher Ω

Plain: one function that knows how to price every kind of pool. *Precise:* `_quoteArms` in the Core is the quote dispatcher for the Solver and for the Quoter's preview. It is private; it is reached through two public doors. `universalQuoteRank(QuoteCtx, amountIn, bar, wantFee)` returns the expected output, a token-denominated depth for the split allocator, and the two figures a ranking may read (`rank` , `hi`); the Solver asks it at its own `bar` , the Quoter at the maximum, because the Quoter reads the output alone. `universalProbe` asks the same arms at `bar = 0` and returns what the Solver's price band needs: the vote's mass, the ranking figure, the fee that figure charged, and two error bounds (§9). A pool the dispatcher 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, so the numbers now run 0 to 9, with seven live kinds and three retired. The numbering keeps its gaps: a Monoslot recorded yesterday must decode identically for ever.

§4.1 Seven kinds, five 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` , after the pair is shown to hold what it declares	reserves · pair-verifiable	90,000
1	`KIND_V3`	Concentrated liquidity (Uniswap V3 and V3-shaped forks)	closed form `outV3` over `slot0` for ranking; the pool's own ticks walked for the promise (§4.2)	concentrated-at-pool · pair-verifiable	110,000

KIND	CONSTANT	FAMILY	PRICING BRANCH	Θ ATTRIBUTES	GAS LADDER (SOLVER)
4	`KIND_V4`	Uniswap V4 singleton	the pool's book walked through `extsload`, tick crossings included	concentrated-in-singleton	180,000
5	`KIND_SOLIDLY`	Solidly (Aerodrome, Velodrome, forks), stable and volatile	ask the pool `getAmountOut`, less one wei; replicated curve as fallback	reserves · pair-verifiable	90,000
6	`KIND_ALGEBRA`	Algebra (Camelot-class), dynamic fee	same branch as kind 1, fee measured from `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
9	`KIND_EXTERNAL`	Delegated: a Gate wrapper (§4.5)	identity first, then the wrapper's own `quote` through `gateQuote`	none, on purpose (Θ = `0x0`)	90,000 (default)

Seven live kinds, five families: constant product {0}, concentrated {1, 6}, Solidly {5}, the V4 singleton {4, 8}, and the delegated wrapper {9}. 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 that read it ("does this kind read reserves?") is one shift and one AND (`kindHas`, `kindHasAny`). A kind whose nibble is zero answers no to every such question and fails closed with no default branch for anyone to forget. The table is unchanged from the 2.3 paper, and kind 9 reads a zero nibble: [derived] reading the constant's ten nibbles from the low end gives 9, A, 0, 0, 4, 9, A, 0, 4, 0 for kinds 0 to 9, and a kind of 10 or more shifts past the 40 bits of the constant and reads zero. The gas column is a second table, `THETA_GAS = 0x2B0016122400001612`, read only by the Solver's `estGas`, one byte per kind in units of 5,000 gas, and a zero byte means the historical default of 90,000; kind 9 has no byte in the constant, so it takes that default.

Kind 9 is an identity, not an attribute. A Gate wrapper speaks the V3 `swap` and `uniswapV3SwapCallback` ABI in front of a curated, code-pinned adapter, so the Router settles it through the same callback and the same measured delta as a V3 pool. But the Core has no reader for the venue's own state (no reserves, no `slot0`), so its Θ field is zero on purpose: no attribute question answers yes, and no code path can treat a wrapper as a V3 pool by accident. What replaces the attribute is the identity check `gateWrapperLive` — the wrapper must derive from the Hub's Gate row and the hop's own tokens and fee — and the price comes from one producer, `gateQuote`. The Hub admits kind 9 only as a `MODE_CREATE2_V3` factory row, and it sits outside `KINDS_EXECUTABLE`, the set the Hub's `recordSwap` accepts; the Router executes it by identity. Kind 8 is never registered by a factory at all: it derives from the singleton.

Kinds 2, 3 and 7 are absent by deliberate excision. They were Curve stable (2), Balancer V2 (3) and Curve crypto (7), retired by the owner's decision of 2026-08-20. Almost no L2 carried them, and they cost bytecode in five contracts; with them went 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. They fail closed in four independent places, all by construction and none by branch: their Θ field is `0x0`; they sit outside

`KINDS_ROUTABLE`, so the Hub's `addFactory` refuses them, and outside `KINDS_EXECUTABLE`, so its `recordSwap` skips them; and the Router's execution dispatch falls into its final `else` and reverts `RouterE(8)` before it touches a pool. The retired numbers are never reassigned: `decodeKind` reads the kind from Monoslot bits, so reusing 2 would reinterpret every pool ever recorded under it. The families themselves have not gone: Balancer and Curve return as data behind the Gate (§4.5), all carried by kind 9, and the three old numbers stay burned.

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 %). That is a dated measurement, not a standing figure.

§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. Before the formula runs, the pair is shown to hold what it declares: `heldBy` compares the two reserves `getReserves` reported against the pair's own token balances, and a pair that declares more than it holds quotes zero, for every consumer (probe, split, leg quote, Quoter). The Solidly arm applies the same check. For a pair that passes, 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 Router's per-leg quote reads the same function (`legQuoteImpact` calls `effV2Fee`), 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 $\text{impactV2Bps} = \lceil x \cdot \text{BPS}/(r_{\text{in}} + x) \rceil = \lceil 10^{14}/1.01 \times 10^{12} \rceil = 100$ 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} (a 128-bit L times a 160-bit $\sqrt{P}$). The Core never forms it — every product is factored through a full-precision `mulDiv` (512-bit intermediate, 256-bit result), so each stage 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).

One interval, and what the promise does about it. The dispatcher's concentrated arm is a single-interval form: it prices the whole order as if the current L continued for as long as the order needs, and every call site of `outV3` passes a zero price limit, so there is no clamp. [derived] Along a swap the marginal price is $dy/dx = P$ whatever L is, because $dy = L\,d\sqrt{P}$ and $dx = L\,d(1/\sqrt{P})$ give $dy/|dx| = \sqrt{P} \cdot \sqrt{P} = P$; what L sets is how far the price must travel to absorb an input. So when liquidity beyond the interval's edge is thinner than L (or absent), the true price falls faster than the form assumes and the form over-estimates the output; when it is thicker, the price falls more slowly and the form under-estimates it. The form is exact only while the order stays inside the interval.

The design does not replicate a venue's tick-crossing arithmetic in the ranking arm, because replicating a venue's internal mathematics is a structural liability: any divergence between copy and original surfaces as a mispriced quote. The ranking uses one convention for every concentrated venue; the *promise* is where the venue's own book is consulted. `v3PaysOnBudget` walks the pool's ticks as the V4 walk does (§5), with a pool identifier of zero and a budget of `V3_PAY_READS = 12` reads of the pool. If the walk finishes inside the reads and inside the walk's step cap, the answer is the smaller of the walked output and the single-interval figure f; if it does not finish, it is the smaller of f and the walked output plus a single-interval estimate of the remainder; a book that does not answer returns f. It never answers above f. Two consumers read it, and it has one producer: the Solver's `_paysWhatItWalks` lowers a leaf's promise when the walk comes in under it, recomputes the hop, leaves the ranking unchanged, and returns the pools whose promise fell to the funnel so the leader is rebuilt without them; the Router reads it through the Core's `legQuoteImpact`, where the impact it reports stays that of one interval, the figure the Solver's floor reads. The Solver's capacity probe takes the same walk for a V3 pool whose holdings cap the order, once per solve topology.

The walk uses the pool's own arithmetic with its own rounding, input rounded up and output rounded down, so it can understate a delivery and never overstate one, including the remainder term of `v3PaysOnBudget`, priced by `outV3`, which takes the pool's own step (the next square-root price rounded up), so it is never a

wei above what the pool pays, and the answer stays capped by f. The promise therefore rests on the pool's own book, not on the one-interval estimate; the Quoter's Exact Pass (§12) gives an integrator the pool's own answer 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 $\sqrt{P}$. A Gate wrapper's depth is the Gate's own `vitalOf` figure (§4.5).

§4.3 Two fee units, carried natively

Constant-product and Solidly venues quote fees in basis points; concentrated venues in parts per million. Each arm computes in its own native unit, which removes a class of rounding failure at the price of two constants instead of one. The only place the two meet is the figure the probe reports back: `universalProbe` returns the fee its ranking figure charged in pips (parts per million), so a V2 or Solidly basis-point fee is multiplied by 100 once, at the edge, and a wrapper, whose fee no read in the Core can see, returns the sentinel `FEE_UNKNOWN` , a value no real fee can take. 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). A V4 pool with a dynamic fee and a protocol fee set together is priced at the fail-closed ceiling instead of at a guess.

§4.4 One dispatcher, three readers

The same arms are read in three places, and the table records what each place asks of each kind.

KIND	QUOTER PREVIEW	SOLVER'S PLAN	ROUTER, IN ITS OWN FRAME
V2 (0)	`outV2` over `getReserves` , after `heldBy`	same	`outV2` through `legQuoteImpact`
V3 / Algebra (1, 6)	one-interval `outV3` over `slot0` / `globalState` ; Algebra's live fee measured from the same read	ranking: one interval; promise: `v3PaysOnBudget` (§4.2)	`legQuoteImpact` : impact of one interval, promise from `v3PaysOnBudget`
Solidly (5)	`solidlyAskOut` : the pool's `getAmountOut` less one wei; replicated curve × 9,800 / 10,000 when the pool does not answer but answers `stable()` (any other pair asks zero)	same	same function for quote and execution
V4 / V4-native (4, 8)	the book walked through `extsload` , fail-closed fee gate	same walk (16 steps for the promise, up to 256 for the ranking)	`_v4LegQuote` → `v4LegOut` : the same walk, fee resolved in the swap's direction
EXTERNAL (9)	`gateQuote` (identity, then the wrapper's `quote`)	`gateQuote`	identity (`gateWrapperLive` , else `RouterE(19)`) and the measured delta; no re-quote

Two details in the table carry the argument. First, the V4 walk is the one place the ranking and the promise differ in depth on purpose: the promise is bounded at 16 steps, so an order that does not finish inside them under-estimates, the safe side; the ranking reads a deeper walk of up to 256 steps, because anyone can place a

one-wei position on the ticks under an honest pool (`modifyLiquidity` has no access control), and an attacker can then only lower a rival's figure, and only with ticks inside the price path of the very order being priced. [derived] At tick spacing 1 each tick moves the price by a factor of 1.0001, so 256 steps cover $1.0001^{256} - 1 \approx 2.6\,\%$; past that the figure is a lower bound and is treated as one. Hooks are not modelled; a dynamic-fee pool whose hook runs before the swap (`beforeSwap`) quotes zero. Second, the wrapper row has no in-frame quote on purpose: the Router asks the wrapper for identity and measures what arrives, and the Router's floors never read a wrapper's quote.

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

§4.5 Venues behind the Gate

Plain: some venues cannot be priced by reading a few storage slots, so the Core does not pretend to. *Precise:* the Gate is one factory row in the Hub in front of venues the Core cannot read natively: Balancer V2, Balancer V3, Curve StableSwap-NG, Curve Twocrypto-NG and Tricrypto-NG, Liquidity Book, PancakeSwap Infinity (CL and Bin) and Algebra Integral. For each venue it creates a thin immutable wrapper at a derived address. To the Router the wrapper looks like a Uniswap V3 pair (kind 9, §4.1): it speaks the V3 `swap` and `uniswapV3SwapCallback` ABI, and it forwards what it measured arriving to its family's adapter. The Hub admits kind 9 only as a `MODE_CREATE2_V3` factory row, so a wrapper can exist in the registry only at the address the Gate's own creation code derives.

What the Router trusts, and what it measures. Nothing a wrapper or an adapter reports about its own output is an input to a floor. The Router measures the balance delta at the recipient, exactly as it does for a V3 pool, and enforces its floors against that. An adapter is immutable, leaves no allowance standing, and after each call may hold no more of either token than before. A venue that can only be priced by simulating it is quoted by `eth_call` ; the quote is advice to the caller, and the Router's execution floors never read it.

How an adapter is admitted, and kept honest. An adapter enters in two steps separated by `ADAPTER_DELAY = 48 hours` . At the proposal the Gate scans the adapter's runtime code opcode by opcode, stepping over `PUSH` data, and refuses `DELEGATECALL` , `CALLCODE` and `SELFDESTRUCT` (code that can run foreign code in its own context, or vanish and be redeployed, would make a pin meaningless), as well as empty code and an EIP-7702 delegation prefix; the scan is conservative and may refuse honest code, never admit dishonest code. The proposal records the adapter's code hash; the acceptance reverts if the code has moved since, and pins the hash. `liveAdapter` is the one producer of "which adapter may run for this family": it answers zero when none is set or when the code at the address is no longer the pinned code, and every swap reads it, so a changed adapter stops every wrapper of its family at the next call. The curator can clear a family at once, which is a brake and not a power over funds: every wrapper of the family reverts from the next call, and the other kinds are untouched. Control can be renounced: afterwards the curator keeps only the grow-only operations, and `familiesUsed` records every family that ever had an adapter, so a family that was live cannot be refilled and its wrappers cannot change hands. Creating a wrapper stays a curator act, because the first creation fixes the venue pool of its (token0, token1, venue key) for ever, and an open door would let anyone occupy every slot with a thin pool of a legitimate venue.

Capacity. A pair holds at most `GATE_MAX_WRAPPERS = 4` wrappers. The reason is a cost, not a taste: the best door reads every wrapper of the pair on every solve, one identity check each, and every wrapper enters the Solver's funnel next to the native venues. Because a wrapper's mass is not measured from a reserve, it is never dropped as dust, so the cap bounds the work the Solver can be handed.

One producer for the price. `gateQuote` is the only function that prices a wrapper, and it asks one question in order. First identity: `gateWrapperLive` requires that the Hub has admitted the wrapper's Gate and that the Gate still runs the code it was admitted with, that the wrapper sit at the address that Gate derives (CREATE2) from this hop's own tokens and fee, and that the family's adapter be live now. Then the wrapper's own `quote(bool,uint256)` is called by `staticcall` under a fixed gas ceiling, `GATE_QUOTE_GAS = 500,000`; a failure is a zero quote and the venue drops out. When the dispatcher asks for depth, it comes from the Gate's `vitalOf`: the adapter's declared depth, capped by the mass the venue's own pool holds, read under twice that ceiling. The Solver and the Quoter's preview both read `gateQuote`; if the Quoter finds a live wrapper whose direct answer is zero, it retries as a dry run in a frame that reverts, which is how a venue priced only by simulation gets a number. The Router, as §4.4 records, does neither: it checks identity and measures the delta.

What it costs in bytecode. The Gate's runtime is 10,710 bytes (13,866 bytes under the EIP-170 limit of 24,576) and the wrapper it deploys is 4,408 bytes (20,168 under); the eight venue adapters run from 3,026 to 6,666 bytes. [measured] The Gate's initcode is 15,865 bytes against the 49,152-byte EIP-3860 limit. All figures are from the release-profile size guard at commit `969bcc6`, CI job 133367307770.

§5 Uniswap V4 end to end

Plain: V4 pools have no address, run other people's code inside the swap, may charge a fee that changes, and may hold the chain's native coin. *Precise:* 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 are entries in one singleton's storage, keyed by a hash. A hook executes at defined points of the settlement the router is performing. A fee need not be constant. And a pool may hold the 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. That 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 three hashes in steps 2 and 4 and the wrapped id below were recomputed for this edition with `cast keccak` over the encoding in step 1. [measured]

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. The layout in step 5 is not taken from documentation alone: the same pool read through this layout and through the canonical `StateView` contract returned the same numbers on a mainnet fork when the reader was written, and that identity is what the Core's comment cites.

The scan. The scan that produces candidate ids has four stages, and none of them depends on another. Before the first cold probe of a pair, one measured read asks whether the PoolManager holds any of each token; if the manager's measured balance of a token is zero, the pair has no V4 pool and the search stops (`_pmHoldsNone`). An absent reading is not a zero: a manager inside an unlock, a failed read or a reply shorter than one word keeps the search going. Then, in order:

STAGE	WHAT IT TRIES	COST WHEN IT HITS
Native pass	the same stages below with c_0 = the native currency; hits are rewritten to `KIND_V4_NATIVE` with token0 = WETH	folded into the stages below
Learned codes	the tier pattern already learned for each token (`v4CodeOf`); equal codes deduplicate	one probe, the steady-state path
Canonical tiers	the four Uniswap tiers (fee, tickSpacing) = (500, 10), (3,000, 60), (10,000, 200), (100, 1), through one batched `extsload(bytes32[])`	one call for the batch
Row extras	the registration row's `fees[i]` paired with `spacings[i]`, never their cartesian product	one batched call

No count stops the scan early, and that is deliberate. An earlier design capped the scan at eight proven pools and probed a generator grid of forty tiers on every miss; a cap lets dust planted in the first cells use the places before a deep pool at a later cell is reached. The cap and the grid are gone. The table's own bound limits what a row emits (a V4 row emits at most `2 · fees.length + 10` candidates per call), the dust that gets through reaches the Solver, whose funnel weighs it against the pair's depth, and a learned hint can speed a search up but cannot suppress one. An exotic tier is reached by a learned code or by a row extra. The generator grid survives only in reverse: `V4_GRID_MAX = 99` bounds `_recoverV4Ts`, which recovers a tick spacing from the code learned out of a swap.

Two properties fall out of the construction rather than out of 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 (hooks are treated in §5.5). And duplicate ids from different tier guesses collapse to one candidate by pool address in `_admitV4`, so one pool cannot occupy the funnel twice by aliasing. `_admitV4` is also where the fee gate of §5.4 refuses a pool whose effective fee reaches 1,000,000 pips.

Probe order is still a policy, because a pair against the chain's native asset has two keys in V4, native and wrapped, and the scan emits both and lets the funnel pick by depth. The native key was the deeper one in every case measured when the ordering was decided: 3.76 times on Base (USDC/ETH, 500/10) and 292 times on Robinhood Chain (USDG/ETH, 100/1). [measured, 2.3; source: the Hub's comment at the scan] Deriving only the wrapped key would never error; it would quote worse, silently.

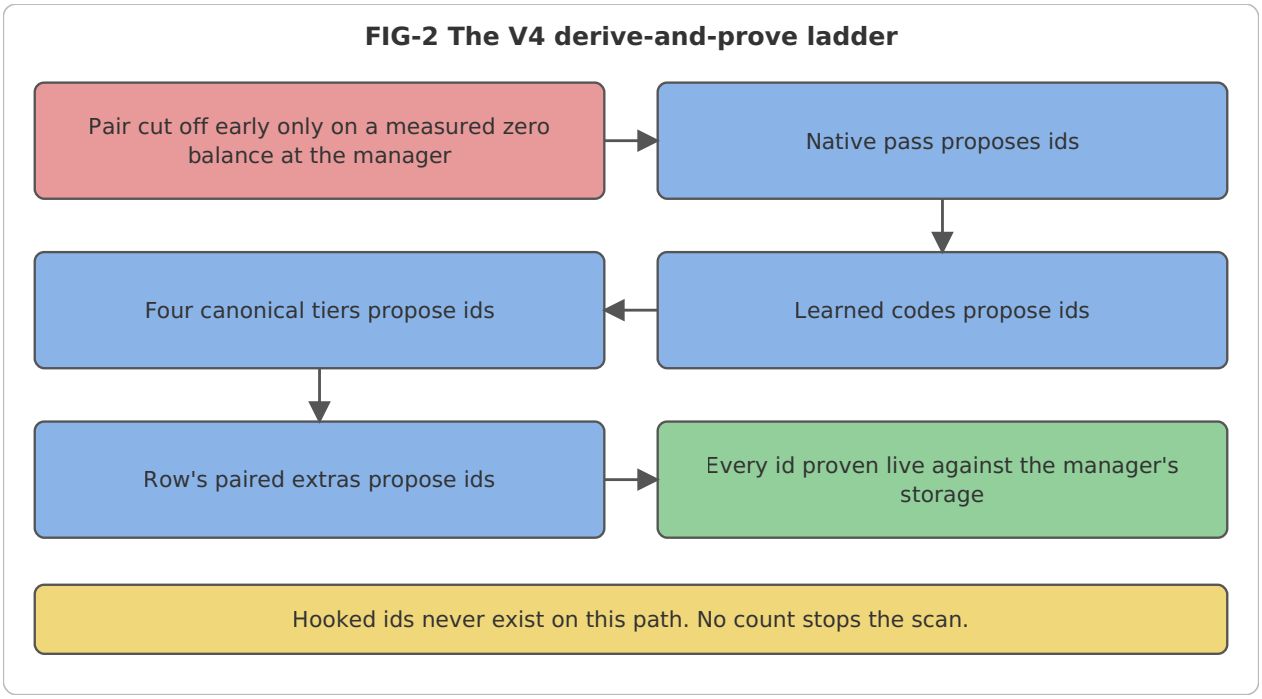

FIG. 2 The V4 derive-and-prove ladder

FIG-2: The V4 derive-and-prove ladder. A pair is cut off early only on a measured zero balance at the manager; otherwise the native pass, the learned codes, the four canonical tiers and the row's paired extras each propose ids, every id is proven live against the manager's storage, and hooked ids never exist on this path. No count stops the scan.

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)`). Four further conditions gate the door. Neither currency may be the zero address: this door takes the wrapped key, and native rows come from the scan above. Claimed entries start provisional and are weighted by measured marginal output. 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 resolve under the fee gate of §5.4. And the claim must clear the same admission margin a swap-driven registration clears (§9), with the pool's measured liquidity, converted to token-denominated virtual reserves on the short side, as its depth. The call is idempotent, and it teaches the learned tier code only if it actually registered a row. 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 worked example 4, and the V4 branch reduces to the concentrated-liquidity closed form of §4.2: the same mathematics, the same overflow discipline, and for the promise the same walk of the pool's own ticks that §4.2 describes. The singleton changes where the state lives, not what the state means. A manager with no code answers a static call with no return data, and `v4SqrtAndLiq` treats a codeless target as all-zero words: fail-closed, never a revert and never garbage read as a price.

§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 the owed currency and take the received one (`sync`, transfer and `settle` for a token; `settle` with value for native), then release the lock. The callback accepts only the manager as caller (`RouterE(6)`). 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 callback clears both slots before it settles anything: a re-entry reads an empty context and reverts. The `BalanceDelta` the manager returns is packed, `amount0` in the high 128 bits and `amount1` in the low, and the Router decodes it as such before any settlement arithmetic runs. A pool that owes the wrong way, or asks for more input than the leg was committed, reverts (`RouterE(8)`): a malicious pool or hook is bounded to the current leg's budget and cannot draw on the input budgeted to its sibling legs.

Native currency keeps the invariant that matters. The Router is WETH-canonical everywhere: native value entering `swapExactInNative` is wrapped exactly once at the door, the route must start in WETH, and the door is fail-closed until `weth` is wired (`RouterE(3)`). Native output is deliberately not implemented: the recipient receives WETH, which no swap path can silently trap. A native V4 leg unwraps just in time inside its own unlock frame (`WETH.withdraw`, then `settle` with value), and ETH received from a native `take` is rewrapped in the same frame, so native ETH never exists at rest in the contract and every balance-delta measurement downstream sees only WETH. 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. Bare ETH from anyone else reverts (`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 in addition take a protocol fee, set per direction by Uniswap's fee controller, and a dynamic-fee pool carries the sentinel `0x800000` in its key while its real fee lives in `slot0`. Quoting the sentinel at face value would price the trade as free while execution paid the live rate; quoting a static key without the protocol fee would overstate the delivered output once a controller enables it. The Core resolves both in one pure function, `effV4Fee`, which takes four arguments (key fee, LP fee, protocol fee, direction) and returns pips:

$$f_{\text{eff}} \;=\; \begin{cases} f_{\text{key}} & f_{\text{key}} \neq \texttt{0x800000} \;\wedge\; p = 0 \\ p + f_{\text{key}} - \lfloor p\, f_{\text{key}} / 10^{6} \rfloor & f_{\text{key}} \neq \texttt{0x800000} \;\wedge\; p \neq 0 \\ \texttt{0xFFFFFF}\ \text{(unquotable)} & f_{\text{key}} = \texttt{0x800000} \;\wedge\; f_{\text{proto}} \neq 0 \\ f_{\text{LP}}\ \text{read from}\ \texttt{slot0} & f_{\text{key}} = \texttt{0x800000} \;\wedge\; f_{\text{proto}} = 0 \end{cases} \tag{E19}$$

Here p is the protocol fee of the swap's direction: the low 12 bits of the packed field for a swap from token0 (`zeroForOne`), the high 12 bits for the other direction. The second case is the composition of two fees charged one after the other on the input. A fraction $p/10^6$ goes to the protocol, so $(1 - p/10^6)$ of the input remains, and the LP fee applies to what remains: the total retained fraction is $(1 - p/10^6)(1 - f/10^6)$, so the effective fee in pips is $p + f - pf/10^6$. The integer division floors the subtracted term, so the figure rounds up by less than one pip, the direction in which a quote can only under-promise. [derived; the composition rule is the one the code's comment cites from Uniswap v4-core's `ProtocolFeeLibrary.calculateSwapFee`] For illustration, $f = 3{,}000$ and $p = 500$ give $500 + 3{,}000 - \lfloor 1{,}500{,}000/10^6 \rfloor = 3{,}499$ pips against an exact $3{,}498.5$. The PoolManager caps p at 1,000 pips (0.1 %) per direction and refuses a larger setting from the fee controller (`ProtocolFeeLibrary.MAX_PROTOCOL_FEE`, `isValidProtocolFee`, Uniswap v4-core).
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 fee a hook may override inside `beforeSwap` is not anchored the way a key fee is. The measured protocol fee is zero on every chain served at the time of the 2.3 edition, so this branch cost nothing in practice. [measured, 2.3] The gate is applied where a pool enters the registry, not only where it is quoted: `_admitV4` and `claimV4` both refuse a pool whose direction-free effective fee reaches 10^6 pips.

The same rule closes the same seam one family over. Concentrated-liquidity forks whose fee lives on the pool are asked the pool first: `quoteV3Fee` asks the pool's own `fee()` before it consults the fee stored in the registry row, and the stored fee answers only for a pool whose `fee()` does not. The order matters because a registered row carries the fee of the block it was written in, while a dynamic-fee pool moves its fee from block to block: on Base, the Slipstream WETH/USDC pool with tick spacing 100 showed 549 and then 564 pips in consecutive blocks. [measured, 2026-10-27] Algebra pools are dynamic-fee by construction and register the zero sentinel, so they keep their own order: the fee measured with the price in the `globalState` read that already supplied it. When nothing answers, the result is `0xFFFFFF` and the quote fails closed. Static V3 pools take no extra read beyond the one staticcall.

One seam is bounded rather than removed: a dynamic-fee hook can override its fee per swap inside `beforeSwap`, so the fee stored in `slot0` 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. The checks are made on the hook's address, on a code hash, and on a list a human keeps, and §14 develops them. Three layers stand, cheapest first.

1. **Address arithmetic.** A hook's permission bits live in the low fourteen bits of its own address, fixed at deployment. The two bits that permit returning swap deltas, `1 << 3` and `1 << 2`, are refused by one AND before a token moves (`hookAltersDeltas`, `RouterE(9)`): a vanilla quote cannot price custom accounting, so quote and execution would diverge. A second AND, `hookRunsInSwap`, asks whether the manager enters the hook during a swap at all: `(uint160(hook) & 0xCC) ≠ 0`, that is `beforeSwap` (`1 << 7`), `afterSwap` (`1 << 6`) and the two delta bits. A hook with none of them is never entered by its own pool's swap, because the manager dispatches on those immutable bits; its pools are admitted by the bits alone, and nothing an operator could judge about its code can change a swap, so they need neither list nor pin.
2. **List and pin, for a hook that runs.** The automatic door proposes a hook only while `isHookLive` holds: it is on the allow-list and its runtime code hash still equals the hash pinned at admission. At execution, a hook that runs in the swap and is paused, whether revoked by the curator or listed and since moved off its pinned code, is refused (`RouterE(9)`). A hook nobody listed is the caller's own signed choice on this door, bounded like every venue by the in-frame promise, the attestation gate and `userMinOut`. The pin does not cover a proxy whose implementation is swapped behind an unchanged code hash; the allow-list is the human judge of that case.
3. **The projector Ξ.** The Quoter returns zero for a hook that alters deltas, or one that runs in the swap and is paused, and the Solver filters delta-altering hooks out of its 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: that is the price of refusing, on principle, to call untrusted code to find out what it would have done. And a hook that does run in the swap can call the manager on any pool, entering that pool's hooks. That reach is bounded by the manager's own flash accounting (a delta left unsettled reverts the whole unlock; measured on a Base fork in `test/fork/NestedHookSettlementFork.t.sol`), by the same floors, and by the admission of the hook that makes the call, never by the bits of the hook it reaches.

§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 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 won more hop-pairs than any other registered venue. [measured 2026-08, private census, Version 2.3]

§6 The Solidly solver

Plain: some pools are priced by a formula too delicate to copy safely, so the protocol asks the pool for its own answer and keeps a careful copy only for pools that cannot answer. *Precise:* some invariants have no closed form simple enough to reproduce without risk, and reproducing them is exactly the replication hazard the protocol refuses elsewhere. For such venues the dispatcher asks the pool, through the narrowest 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 stays closer to the diagonal than constant product while still admitting divergence:

$$k(x,y) \;=\; x^3y \;+\; x\,y^3 \;=\; x\,y\left(x^2+y^2\right) \tag{E5}$$

At the balance point the slope of the level set is -1, as for constant product: $\mathrm{d}y/\mathrm{d}x = -(3x^2y+y^3)/(x^3+3xy^2) = -4x^3/4x^3$ at $x=y$. What differs is the curvature, and the figure below plots it. A zero-fee pool with $X=Y=R$ and an input of 1 % of R returns 99.99995 % of the input; constant product returns 99.0099 %. At 10 % the two return 99.950 % and 90.909 %. [derived: exact rational bisection of $k(R+a,R-\text{out})=k(R,R)$, 200 steps; constant product is $a\,R/(R+a)$] Volatile Solidly pairs are constant product, and the Core's volatile arm *is* `outV2`: one producer, which metamorphic relation MR7 pins (§15).

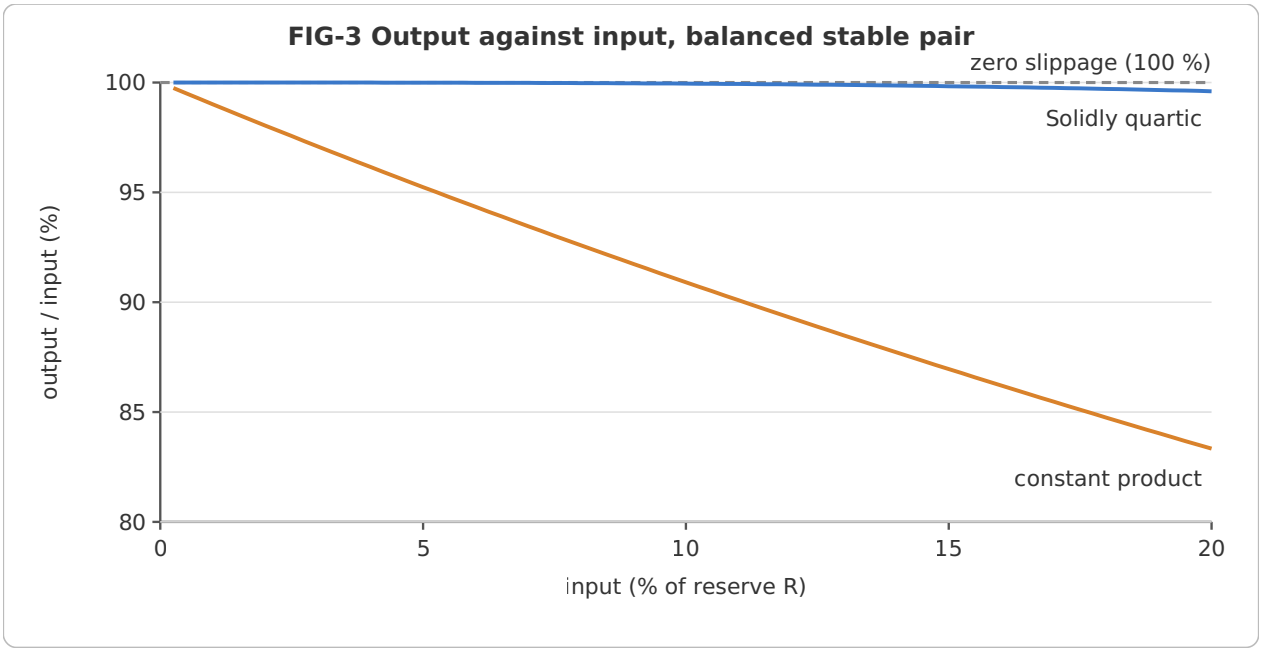

FIG. 3 Output against input, balanced stable pair

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.

§6.1 Primary path: ask the pool

`getAmountOutWithTotalFee(uint256,address)` (selector `0x0ea099c1`) is asked first and, only when that ask fails, `getAmountOut(uint256,address)` (`0xf140a35a`). The answer is computed by the pair's own bytecode, with the live fee, curve and rounding it will apply at swap time, so a swap that requests no more than that figure sits inside the pair's own invariant check. The Core asks for that figure less one wei, the rounding margin that keeps the pair's K check satisfied; no proportional haircut applies on this path. [checked in code: C:1470-1490, 1516-1519, 1531-1536] [selectors recomputed with `cast sig`, 2026-11-16]

Every such ask is bounded in the way every read the quote path makes of a pool is bounded: at most `GAS_CAP` = 100,000 gas, a fixed-size copy of one word, never the whole returndata, and a floor of `ASK_GAS_FLOOR` = 104,587 gas below which the ask reverts empty instead of answering "no". A starved ask therefore cannot pass for a pool that has nothing to say. [checked in code: C:215, 223, `_askWord` (C:1497) and its comment]

The trigger for the fallback is an answer of at most 1 wei, or no answer at all. It is the same test in the three channels that ask (the quote dispatcher that serves the Solver and the Quoter, and the Router's execution), because all three reach one function, `_solidlyAsk` (the Router through its public wrapper `solidlyAskOut`), rather than each carrying a copy: the Router's call site says "the same producer the quote channel promises", and an earlier divergence, one channel falling back on an answer of exactly 1 while another did not, is the reason it is one producer. [checked in code: C:1516 and 2012, R:1983-1984; the earlier divergence is from the 2.3 text and is not re-derived here]

§6.2 Fallback: replicate, then under-ask

Only a pair that answers `stable()` (selector `0x22be3de1`, an ABI bool: a word above 1 is no answer) has a curve to replicate. For any other pair the `stable` flag is nobody's word: the volatile curve on a stable pool under-asks a balanced pool and over-asks a skewed one, so the ask is 0 and the venue is surrendered to the floors. When it does answer, the dispatcher reads the live fee, solves the invariant on decimal-normalised reserves (§6.3, §6.4) and multiplies the result by 9,800 / 10,000, a 200 bps haircut (`SOLIDLY_CURVE_ASK_BPS`), so that the pair's rounding, which the replica cannot observe, always has slack. The haircut is a property of the request and not of the curve: it applies on the fallback and never when the pool answered for itself. [checked in code: C:213, 717-722, 1516-1541]

The live fee comes from a short ladder, and the ceiling applies to one rung only.

RUNG	SOURCE	RULE
1	the pair's `factory()` then `factory.getFee(pool, stable)` (selector `0xcc56b2c5`)	believed when $0 < \text{fee} < 10{,}000$ bps; no ceiling
2	the fee declared with the leg (calldata on the Router's doors)	believed only when it is at most 100 bps; 0 or above 100 bps is replaced by rung 3
3	a fixed default	30 bps

RUNG	SOURCE	RULE
lie	`factory()` answers a word that is not an address	fee = 10,000 bps: the pair prices at zero and the pair's other venues plan as before

The ceiling exists because a Solidly-shaped pair with no `getAmountOut` and no working factory leaves the declared fee as the only fee source: an over-declared 99 % would collapse the leg's on-chain quote to about 1 % of the truth, and when that leg is the last hop it would drag the protocol floor down with it. The V2 arm closed the same hole with the same constant (`V2_FEE_CEILING_BPS` = 100), and the Solidly arm now reads through it. A `factory()` of zero (no contract behind the call) keeps the declared or default fee. [checked in code: C:1327, 1747-1778]

The haircut is also a charge, and the price band must see it as one. The effective fee of the fallback ask, in pips (millionths), is $1 - (1 - f) \cdot 9800/\mathrm{BPS}$, written in the code as `1_000_000 - (BPS - fee) * 9800 / 100`. For the default f = 30 bps this is $10^6 - 9970 \cdot 98 = 22{,}940$ pips, 2.294 %. [derived from C:1541 and C:1522-1526; arithmetic checked]

§6.3 The solve

Given the post-trade input balance x and the target invariant K from the pre-trade reserves, the Core seeks the root in y of $f(y) = x^3y + xy^3 - K$, with $f'(y) = x(x^2 + 3y^2) > 0$ for $x > 0$ (`_solY`, with f' computed as `fp`). The seed is $y_0 = Y$, the opposite pre-trade reserve. Two facts make that the right place to start. The root cannot lie above Y: k is increasing in x, so $k(X + A, Y) \geq k(X, Y) = K$ and the root of f is at or below Y. And $f'' = 6xy > 0$, so f is convex and increasing, and Newton's method started to the right of the root of a convex increasing function stays to its right and converges monotonically. [derived; both statements are exact-arithmetic facts, and the integer code adds the guards below for rounding]

$$k(x,y_n) < K:\ \ y_{n+1} = y_n + \max\left(1,\ \left\lfloor \tfrac{K - k(x,y_n)}{f'(y_n)} \right\rfloor\right); \qquad k(x,y_n) \geq K:\ \ y_{n+1} = y_n - \delta_n,\ \ \delta_n = \begin{cases} \lfloor y_n/2 \rfloor & \max(1,d_n) \\ \max(1,d_n) & \text{otherwise} \end{cases} \tag{E6}$$

In the code k and f' are carried in WAD fixed point: `_solK` evaluates $a = xy/\mathrm{WAD}$, $b = (x^2 + y^2)/\mathrm{WAD}$ and $k = ab/\mathrm{WAD}$, each through a 512-bit `mulDiv`, and `fp` is $x(x^2 + 3y^2)/\mathrm{WAD}^2$, so the step $(K - k) \cdot \mathrm{WAD}/f'$ is the real-number step of the formula above. [derived: units checked, $(\Delta K/\mathrm{WAD}^3) \cdot \mathrm{WAD}^3/(f'/\mathrm{WAD}^2)/\mathrm{WAD}^2$ reduces to $\Delta K/f'$]

Every step is bounded. A downward step that would cross zero is replaced by half the iterate; every step forces at least one unit of progress; a vanishing derivative, $x = 0$ or $K = 0$ returns the seed. The iteration exits when two successive iterates differ by at most one unit, and then lands on the side of the invariant that rounds against the taker: it climbs y one wei at a time while $k(x, y) < K$, and gives up after two climbs; then it steps y down one wei at a time, at most twice, while the wei below still holds $k(x, y - 1) \geq K$, so a last downward step that stopped one wei above the optimum is recovered and the exit stays on the safe side. The loop is capped at 64 iterations. Both the cap and the give-up fail closed in the same way: the solver returns the seed itself, equal to Y, so the caller's guard $y \geq Y$ maps it to a zero quote and the pool is treated as unpriceable. Returning zero there would make the caller compute out $= Y - 0 = Y$, the whole output reserve, which is why the failure value is the seed and not zero. [checked in code: C:1832-1883 for every clause above; the "two climbs" figure is the code comment's, "two climbs covered every sampled exit"]

Landing on $k(x,y) \geq K$ matters because $\text{out} = Y - y$: a larger y is a smaller payout, so an exit with $k(x,y) \geq K$ never promises an output that would leave the invariant below its pre-trade value. [derived from k increasing in y for $x > 0$; the integer floors can make k flat across a step of WAD/x wei, which is what the climb handles, per the code comment]

§6.4 Decimal normalisation

The invariant is homogeneous of degree four: scaling both reserves and the input by λ scales k by λ^4 and the solution y by λ, so a quote is scale-invariant in exact arithmetic. [derived: $k(\lambda x, \lambda y) = \lambda^4 k(x,y)$] Integer arithmetic is not scale-free, and a pair such as DOLA (18 decimals) against USDC (6) would carry its mismatch into the quartic, so the ask path always evaluates on a common 10^{18} basis and de-scales afterwards. With $s_{\text{in}} = 10^{18-d_{\text{in}}}$ and $s_{\text{out}} = 10^{18-d_{\text{out}}}$:

$$X = r_{\text{in}} \cdot s_{\text{in}}, \quad Y = r_{\text{out}} \cdot s_{\text{out}}, \quad A = \left\lfloor \frac{x \cdot s_{\text{in}} \,(\text{BPS} - f)}{\text{BPS}} \right\rfloor, \quad \text{out} = \left\lfloor \frac{Y - y^{\star}(X + A,\, K)}{s_{\text{out}}} \right\rfloor, \quad K = k(X,Y) \tag{E7}$$

The fee is taken from the input before the solve (A already nets it), and the output is floored when de-scaled. [checked in code: C:1722-1738] Two entries reach this arithmetic. `outSolidlyStable` takes the decimals explicitly; `outSolidly`, the equal-decimals entry, passes 18 for both sides, which is the identity scaling, and returns the volatile `outV2` for a pair that is not stable. The ask path (`solidlyCurveOut`) takes the real decimals of the input token and of the other token: the other token is identified from the pool's own `token0()` and `token1()` and not from the caller, because a single-producer primitive that accepted its caller's coordinate would reopen the divergence it exists to close. Decimals come from `decimals()` under the same bounded read as every other ask; an answer of 78 or more, or no answer, is read as 18, and a token reporting 19 to 77 decimals quotes zero rather than underflow `10 ** (18 - d)`. A 0-decimal token is a real value and is scaled by 10^{18}; an earlier form that read zero as "already normalised" no longer exists. [checked in code: C:1544-1560, 1655-1681, 1683-1700 (comment "NO SENTINEL"), 1048-1070 (`_decimalsOf`)]

§6.5 Fail-closed on the whole domain

Every test that decides whether the curve is evaluated at all returns zero rather than reverting, because a revert inside a quote is a library `DELEGATECALL` unwinding the Solver's whole plan for the pair: one unquotable pool must cost the pool and not the pair. In the order the code asks them, all on the normalised values of §6.4:

1. A token reporting more than 18 decimals quotes zero; a zero input, a zero reserve or a fee of 10,000 bps or more quotes zero.
2. X and Y must each be at most 3.4×10^{38}, and $X + A$ must be too. The constant is the decimal literal `3.4e38`, which is 2^{128} to two figures ($2^{128} = 340{,}282{,}366{,}920{,}938{,}463{,}463{,}374{,}607{,}431{,}768{,}211{,}456$), the largest value whose square fits a 256-bit word. [derived]
3. `_solKFits(X + A, Y)` must hold: both the invariant $k = xy(x^2 + y^2)/\text{WAD}^3$ and the derivative $x(x^2 + 3y^2)/\text{WAD}^2$ must fit a word at the seed, which the code comment names the largest point the iteration sees, so that one check covers the loop. [checked in code: C:1731-1735, 1801-1809; the "largest point" statement is the code's comment]

The first constant is a sentinel and the third test is the real bound, and the difference is quantitative. For a balanced pool $x = y = Z$, $k = 2Z^4/10^{54}$, which leaves 256 bits when $Z^4 > 2^{255} \cdot 10^{54}$, that is for $Z > 4.905 \times 10^{32}$ in 18-decimal units, about 4.9×10^{14} tokens per side. [derived] Between 4.9×10^{32} and 3.4×10^{38} the sentinel alone would admit a pool and `mulDiv` would revert inside the computation; the code comment on the constant records exactly this, and `_solKFits` is the test written in the term's own units rather than in reserves.

`_solKFits` asks its question without a division, which is what lets it be asked on dust. `mulDiv(a, b, WAD)` reverts exactly when the high word of the 512-bit product $a \cdot b$ is at least WAD: writing $ab = h \cdot 2^{256} + l$, the quotient fits a word iff $h < \mathrm{WAD}$. `_mulFitsWad` computes h with the standard identity $(\mathrm{mulmod}(a, b, 2^{256} - 1) - (ab \bmod 2^{256}))$, less one on borrow) and compares it with one WAD; two multiplications, no division, no overflow on any input. [derived; checked: C:1785-1789] The earlier form, $a \le \max \cdot \mathrm{WAD}/b$, 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, and a whale input reserve against a dust output reserve, which passed the k test and overflowed the derivative on the first step, is why the derivative is asked as well. The repair went in red-first, with three named tests, 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; it is 35 bytes smaller than the code it replaced. [checked in code: ASSURANCE.md section 4n (N-version), lines 700-720, VERIFICATION-REPORT-2026-09.md:59]

§6.6 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, replica output taken before the haircut:

POOL	STATE	GROUND TRUTH	REPLICA 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 %

[measured against live Aerodrome pools on Base before 3 August 2026; first published as Table 6 of the previous whitepaper] The deviations are recomputed here from the three pairs of figures: 0 and $+0.2506$ % and $+0.2492$ %. [derived] The balanced pair matches to the integer; the skewed pairs sit about a quarter of a percent above the truth, on the side that would over-promise, because the pool's post-fee balance rounding is invisible from outside. That is why the fallback under-asks by 200 bps: after the haircut the three rows become 979,543,389, 968,238,320 and 883,534,513, which are -2.000 %, -1.754 % and -1.756 % of the truth, so the margin is about eight times the largest residual measured and the replica's over-promise cannot reach the pair's own check. [derived: $200/0.2506 = 798$ per cent of the residual; one measurement set of three pools, not a bound]

The margin is applied identically on the preview and in-frame surfaces, since both reach it through the one function `_solidlyAsk` (the Router through its wrapper `solidlyAskOut`), so the number a trader is shown is the number the Router requests, and the planner ranks the venue on that same figure. The cost of the haircut is therefore visible in the ranking and in the fee the band sees (§6.2), not hidden in execution. The canonical-oracle lane of §15 bounds the curve independently: against an implementation of the venue's published in-

variant solved by Newton's method, the Core never exceeds the curve by more than the solver's own last step, and measured within 4 wei over 5,000 fuzzed inputs. [checked in code: ASSURANCE.md section 4k table]

§7 Deterministic Derivation $\mathcal{D}$

Plain: before a pool can be quoted it has to be found, and there are two ways to find it: ask the factory, or work the address out. An answer from a factory is a claim; an address worked out is a result anyone can check. The protocol works the address out wherever the arithmetic allows and asks only where it does not. *Precise:* 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, with four factory-call modes for 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 holding the same three inputs; an address that is fetched can only be trusted.

§7.1 The mechanism and the table

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 preimage is $1 + 20 + 32 + 32 = 85$ bytes (0x55) [derived]. The Core spells it out exactly once, in `create2Address`, which hashes the 0x55 bytes at its memory offset and masks the result to twenty bytes; `deriveAddress` selects the salt and the origin from the mode, and the Hub's attested Algebra derivation (§7.3) calls the same producer. One producer means the live derivation and the attested one cannot drift apart.
The table below is the whole resolution surface. The selectors are recomputed with `cast sig`, 2026-11-16.

MODE	FAMILY	RESOLUTION	SALT OR PROBE DETAIL
0	V2	factory call `getPair(address,address)`, selector `0xe6a43905`	none
1	V3 / Algebra	factory call `getPool(address,address,uint24)`, `0x1698ee82`; on an empty answer, `poolByPair(address,address)`, `0xd9a641e1`	none
2	Solidly	factory call `getPool(address,address,bool)`, `0x79bc57d5`; on an empty answer, `getPair(address,address,bool)`, `0x6801cc30` (classic Velodrome V1 shape)	none

MODE	FAMILY	RESOLUTION	SALT OR PROBE DETAIL
3	Concentrated Solidly forks (Slipstream, Velodrome CL)	factory call `getPool(address,address,int24)`, `0x28af8d0b`	none
4	V2 CREATE2	pure arithmetic	salt `keccak256(abi.encodePacked(t0, t1))`; origin = the factory
5	V3 CREATE2	pure arithmetic	salt `keccak256(abi.encode(t0, t1, fee))`; `fee == 0` is the Algebra sentinel: salt `keccak256(abi.encode(t0, t1))`, origin = the factory's `poolDeployer()` (`0x3119049a`), attested at admission
6	Solidly clone CREATE2	pure arithmetic	salt `keccak256(abi.encodePacked(t0, t1, stable))`; origin = the factory
7	Concentrated-liquidity CREATE2	pure arithmetic	salt `keccak256(abi.encode(t0, t1, tickSpacing))`; origin = the factory
8	retired	a tombstone left by the Curve excision: `MODES_VALID = 0x2FF` refuses it at admission, and discovery stops on it without producing a hit	none
9	V4 derive-scan	derive hookless poolIds and prove each live on the PoolManager (§5.1, §7.5)	learned codes, four canonical tiers, the row's paired extras; every candidate proven live

`t0 < t1` throughout: `deriveAddress` sorts the pair before use. Modes 0-3 are one `staticcall` each (two when the dialect fallback of §7.3 fires); modes 4-7 are zero calls and need only the correct init-code hash; mode 9 serves the one family whose pools can be located neither way.

§7.2 A theorem, a claim, and a proof

A derived address is a theorem over the pair: given the origin, the salt 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. So an asked pool (modes 0-3) proves its own pair before discovery lists it. In the probe, after the candidate is known to carry code, the Hub reads the pool's `token0()` (selector `0x0dfe1681`) and `token1()` (`0xd21220a7`) and compares each full 32-byte word with the sorted pair; any difference drops the candidate. The reads are the same two the Router makes at the seam that pays, so a pool the executor would refuse is never listed, planned or ranked. [checked in code: H:1238-1240, C:1377-1395]

Three properties follow from that shape, each readable in the code and none a claim about the venue. The proof runs on asked pools only, because a derived address is already a function of the pair. A pool whose `token0()` fails, runs out of gas under the cap, or answers fewer than 32 bytes yields the zero address, which differs from every real token, so the candidate is dropped: the check fails closed. And the comparison is against the Hub's own sorted pair, not against anything the factory returned. [derived from H:1240 and C:1377-1395]

The hostile-venue matrix of §15 is what asked for this proof. It holds one misbehaviour per venue, among them a factory whose `getPair()` answers with a pool on other tokens, and crosses each with the calldata door and the solve-in-transaction door: settle with the delivered amount equal to the balance delta and nothing left on the Router, or refuse with a selector of ours. Its first run listed the impostor pool and the planner ranked it; the executor refused it at the seam that pays, so funds were never at risk, but the pair was refused while the impostor won the split. The pair proof at discovery was added to close that (2026-09-05). [measured, ASSURANCE.md section 4i; the dated commit is 917f1ea, 2026-09-05]

§7.3 Two subtleties in the V3-shaped rows, and one in the asked ones

Algebra's origin. Algebra (Camelot-class) charges a dynamic fee, so a fee can never be part of its identity. An Algebra registration therefore declares every fee as the zero sentinel (admission refuses a non-zero fee on an Algebra row in mode 5), mode 5 drops the fee from the salt, and the CREATE2 origin is the separate `poolDeployer()` contract Algebra factories expose rather than the factory. That origin is the one derivation input a factory could still steer after admission: a proxy changes its answer while its code hash stays intact. So the Hub asks `poolDeployer()` (selector `0x3119049a`) once, at `addFactory`, stores the answer as `factoryDeployer`, and derives from the stored origin ever after; a zero stored value means the factory itself is the origin. After control is renounced the attestation is frozen: a row already admitted cannot take a first pin, an existing pin never follows the live answer, and a resolver that stops answering never demotes a good pin to zero. [checked in code: H:904-933 (mayPin), H:1215-1230, C:678-692]

The dialect fallbacks. Two families spell the factory lookup differently from the canonical selector. Algebra factories expose `poolByPair(address,address)` rather than `getPool`, so a mode-1 lookup that returns nothing retries under `0xd9a641e1` before giving up. Classic Solidly factories (Velodrome V1) expose `getPair(address,address,bool)` rather than `getPool`, so a mode-2 lookup that returns nothing retries under `0x6801cc30`. A factory that answers the canonical selector never pays either retry, and each ask is one `staticcall` under `GAS_CAP` = 100,000 gas that accepts exactly one 32-byte word. [checked in code: C:745-780, C:215]

The asked rows are pinned by code, then proved. A factory-call row (modes 0-3) depends on what the factory answers later. Two independent checks stand between that answer and a listing. At admission the Hub stores the factory's code hash, and discovery skips a factory-call row whose factory no longer carries it; the skip fails closed and never reverts, so a changed dependency stops producing candidates while every other factory keeps serving the pair. And every answer that does come back is proved against the pair (section 7.2). The first check watches the factory's code, the second watches what it returns; neither relies on the other. [checked in code: H:891, H:1162, H:1240]

§7.4 The final gate, and where deriving stops

Every candidate passes one gate. The Hub discards any derived or asked address that carries no runtime bytecode. `extcodesize > 0` 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: before a hit is stored, the probe compares its address with every hit already stored. Admission also closes the other end: a CREATE2 mode with a zero init-code hash is refused (HubE(5)), and discovery repeats the test, so the derivation stays closed if admission ever changes. [checked in code: C:837-841, H:1210, H:1234, H:1243-1247, H:777-783]

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 the public inputs, so the protocol asks. [carried from the 2.3; the table's modes 2 and 3 carry that choice, C:745-775] Uniswap V3 is the case in the other direction: its init-code hash is one constant wherever the canonical deployment is used, and the fork lane registers that one constant by derivation on Base, Optimism and Arbitrum, where routes found through the derived addresses are previewed and executed against live liquidity; Robinhood Chain's V3 is a separate deployment, so there the protocol asks the factory (§17). Derive where the arithmetic is sound, ask where it is not, and verify everything either way.

§7.5 Discovery consumes the table row by row

For each of at most `MAX_FACTORIES` = 16 registered factories, `discoverFor` runs one scan, and the scan's worst case is fixed by the registration, not by the chain's mood. A known factory is refreshed in place and never given a second row, so one address cannot fill the sixteen seats. [checked in code: H:144, H:803-818] The grid a row probes is its fee cells by its spacing cells, and an axis the row's mode never reads collapses to one cell: modes 1 and 5 read the fee axis, modes 3 and 7 read the spacing axis, and a mode that reads neither (0, 2, 4, 6) probes a single cell. Solidly rows (modes 2 and 6) probe each cell in both flavours, stable and volatile. The ceiling that `discoverFor` allocates is the sum over rows of cells × flavours, and it is computed by the same helper (`_axes`) that the scan loop uses, so the allocation and the loop cannot drift apart. [checked in code: H:285-286, H:1108-1117, H:1164-1180, H:1192-1198]

Mode 9 is bounded by its table too, not by a count. A V4 row is scanned in up to four stages, and no stage is gated on the hits of another:

STAGE	WHAT IS PROBED	PROBE COST
native pass (only when the pair contains the wrapped native token)	the pair with the native currency in place of the wrapped one: the four canonical tiers (fee/spacing 500/10, 3000/60, 10000/200, 100/1), then the row's paired extras	one batched read each
learned codes	the packed tier code learned for each of the two tokens (identical codes probed once)	one probe each, warm in the steady state
canonical tiers	the same four tiers for the pair as given	one batched read

STAGE	WHAT IS PROBED	PROBE COST
the row's extras	`fees[i]` paired with `spacings[i]`, never a cross-product; admission requires equal lengths	one batched read

Every candidate is hookless: its poolId is recomputed from `(currency0, currency1, fee, tickSpacing, hooks = 0)` and is emitted only if the PoolManager's own storage shows it live (nonzero sqrt price and nonzero liquidity, read by `extsload`). A hooked pool cannot be emitted, because its hookless poolId does not exist in the manager. A learned code is a hint and never suppresses a search: the table is probed in every case, so a code planted by anyone cannot switch off a stage. [checked in code: H:1257-1344, H:1428-1448, H:1350-1405]

The worst case for a V4 row with n paired extras follows from that table: the native pass and the wrapped pass each probe $4 + n$ tiers, and the two learned codes add at most two, so the row can contribute at most

$$2\,(4 + n) + 2 \;=\; 2n + 10$$

pools [derived]; the allocation in `discoverFor` is `2 * fac.fees.length + 10`, the same expression. The 2.3's "a V4 row reserving up to eight" is replaced by this: the reservation grows with the registered extras instead of capping at a constant, and a cap that dust at early tiers could exhaust is gone. The configuration surface is guarded at the door (§9): a structurally impossible venue cannot be stored.

§8 The Monoslot

Plain: the registry keeps everything it knows about a pool in one 256-bit word, so remembering a pool costs one storage read and one storage write, not a dozen. *Precise:* 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 routing state into a single word, the Monoslot, read once and decoded by pure arithmetic. [checked in code: C:2813-2866]

BITS	FIELD	WIDTH	MEANING
0	`active`	1	slot is live
1–7	reserved	7	bit 5 = stable curve (measured 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 (§9.2)
8–31	`fee`	24	fee tier, in the venue's native unit; 0 is the sentinel for "measure the live fee"
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; packed but not read by the score
60–63	`bucket`	4	logarithmic depth bucket

BITS	FIELD	WIDTH	MEANING
64–95	`lastUpdateTs`	32	wall-clock timestamp of the last anchored tick
96–127	`emaIn`	32	reserved; written as zero at registration, not updated by the shipped tick
128–159	`emaOut`	32	reserved; as above
160–191	`swapCount`	32	activity count, saturating at $2^{32}-1$
192–223	`regBlk`	32	registration block
224–255	`tau`	32	the pool's own clock: a moving mean of the seconds between its anchored ticks; 0 = not yet measured

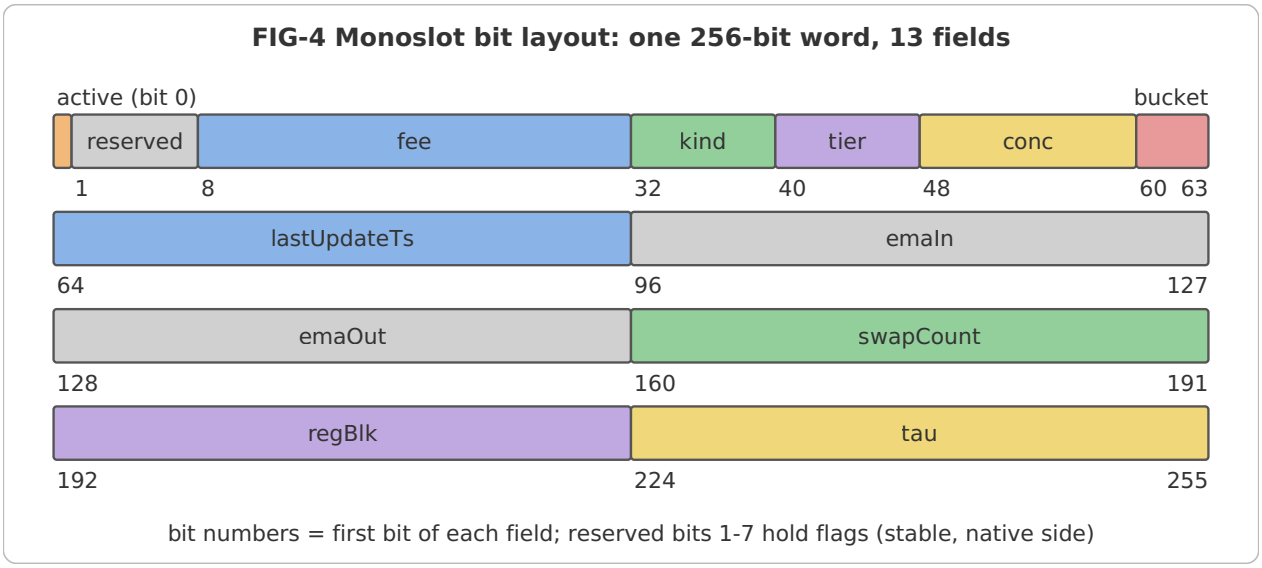

FIG. 4 Monoslot bit layout: one 256-bit word, 13 fields

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

The top field changed after the 2.3. It used to carry the block of the last touch, written on every tick and read by nothing; a block is a different amount of time on every chain, so the field now carries `tau`, the pool's own measured rhythm, which the score reads (§9.2). The encoding is still a sequence of shifts and ORs (`encodeSlot`, with the depth bucket written by `setBucket`):

$$s = a \mid f \ll 8 \mid k \ll 32 \mid r \ll 40 \mid (c \wedge \texttt{0xFFF}) \ll 48 \mid b \ll 60 \mid t \ll 64 \mid e_{\text{in}} \ll 96 \mid e_{\text{out}} \ll 128 \mid n \ll 160 \mid B_{\text{reg}} \ll 192 \mid \tau \ll 224 \tag{E9}$$

[checked in code: C:2840-2867; the term $B_{\text{last}} \ll 224$ of the 2.3 is replaced by $\tau \ll 224$]

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. [checked in code: C:2860, C:2885-2890, C:3286-3319] The `tickSlot` writer clears the count, `tau` and timestamp fields together and writes the three back, so a tick cannot leave a half-updated clock. [checked in code: C:3286-3319]

Notice what the word does not contain: a price. The Monoslot carries what is 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 cannot price a trade. Prices are computed fresh, from live reserves, inside the frame that trades on them; the packed word only decides which pools are worth asking. [derived from §4.4 and §9.6: no quote path reads a Monoslot field as a price] The update path honours the same economy: a swap through a known pool rewrites the word once, one `SSTORE` over one `SLOAD`. The bucket is re-computed on every tick; the clock fields (count, `tau`, timestamp) move only on an *anchored* tick, one whose smaller side moved at least 2^{-16} of the pool's mass (`MATERIAL_SHIFT` = 16), so dust pays gas and moves no clock (§9.2). [checked in code: H:1796-1813, C:2932-2936, C:3234-3247, C:3286-3296]

Why one word matters is a question of multiplication. For each pair the Solver scores the registered pools of up to four topologies: the direct one and one per bridge slot (`TOPOLOGIES = 1 + BRIDGE_SLOTS` = 4, with `BRIDGE_SLOTS` = 3, the Hub's `MAX_BRIDGE_ROUTES`). [checked in code: S:113-114] The 2.3 stated a per-pair candidate cap of eight and "about thirty pools" scored per trade; no `MAX_CANDIDATES` constant exists in the Solver, Hub or Core at the commit this paper cites, so the cap and the product are not restated. The ceilings the code names are per pair and per route: at most `MAX_SLOTS` = 16 registered pools per pair (§9.3), at most four Gate wrappers per pair, and at most `MAX_LEGS_PER_HOP` = 5 legs on each of at most `MAX_HOPS` = 2 hops. At 2,100 gas per cold slot, a five-slot-per-pool layout would price a scoring pass of N pools at $5N \times 2{,}100$ gas, and the Monoslot at $N \times 2{,}100$: a fivefold saving that grows with N [derived: the five-slot layout is a counterfactual, not a measured design]. The Hub's `psisOf` returns every candidate's score in one batched call. [checked in code: H:1672] The Monoslot is not an optimisation applied to on-chain routing; it is one of the reasons on-chain routing is affordable.

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

Plain: the registry learns which pools are alive by watching them trade, and forgets the ones that go quiet; nobody has to feed it a list. *Precise:* 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`: kinds 0, 1, 4, 5, 6 and 8), a local defence that does not depend on the Router staying what it was at deploy, and then takes one of two paths. [checked in code: H:1796, H:209-215]

A known pool's Monoslot is ticked, once, through `tickSlot`. Before that, a row that lies is not refreshed: the Hub asks the *row's own* kind whether the pool still holds what it declares, and a pool that now declares more than it holds stops ageing in, so its score decays and `_canInsert` takes the seat back. The swap itself already executed and is unaffected. [checked in code: H:1810-1813] The tick re-buckets depth every time; the count, `tau` and timestamp move only on an anchored tick (§8), counted from the *currently decayed* base, never from the raw historical total. [checked in code: C:3286-3319] An unknown pool takes the cold path. It must first clear admission (`_canInsert`, §9.3). 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, so without this proof an attacker could register 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, 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. [checked in code: H:1845-1895]

What the row stores is also measured, not declared. The `fee` and `hooks` that arrive in calldata do not enter the row: the hooks field is written as zero (every path to this point has either proven the pool hookless or belongs to a hookless kind); the fee is the authenticated one for V4, the sentinel 0 for Algebra and for a V3 pool whose live fee must be read, and for V2 and Solidly (which have no `fee()`) also 0, so the house's single fee producer answers instead of the caller; and the *kind* is refuted by the pool's own shape (`provenShape`), because a kind planted by whoever front-runs the first swap would otherwise be read as the pool's shape for ever. [checked in code: H:1898-2032 (the comment block REG-01/REG-02 and the derive-don't-declare rule), C:3025] This is the "coordinates, not facts" rule of §2.4 applied to the registry's own door.

The pair-proof at registration dates from September 2026. The Hub campaign offers the Hub pools trading other tokens 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 (§15.3). 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): the pool must pair with a routable bridge, show a nonzero price and nonzero liquidity in the manager's own storage, and be quotable at its fee (`HubE(9)` otherwise). [checked in code: H:1036-1054] Both doors 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`):

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

[checked in code: C:2961-2973] 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. [checked in code: H:2112-2114]

The decay, on the pool's own clock. In the 2.3 every pool decayed by one fixed step, 24,576 s. That treats a pool that trades every minute and a pool that trades every week the same way: the busy one keeps a stale row for days, and the weekly one dies between two honest trades. Each row now carries τ, a moving mean of the time between its own anchored ticks (§8), and the decay runs in steps of $h=4\tau$ (`VITALITY_STEPS_PER_TAU` = 4); while τ is unmeasured the step is the old 24,576 s. [checked in code: C:2925-2926, C:2985-3004] With age $a=t-t_{\text{last}}$:

$$h \;=\; \begin{cases} 4\tau & \tau>0 \\ 24{,}576\ \text{s} & \tau=0 \end{cases} \qquad \sigma=\lfloor a/h\rfloor, \qquad c(s,t) \;=\; \text{swapCount}(s)\gg\sigma \;-\; \left\lfloor\frac{c\,(a \bmod h)}{2h}\right\rfloor, \qquad V=\max\!\left(1,\ \left\lfloor\frac{c\cdot 24{,}5}{h}\right.\right. \tag{E11}$$

and the pool is dead ($V=0$) once $\sigma>31$. [checked in code: C:2945-2959, C:2985-3004] Inside a step the count falls along the chord from c to $c/2$ instead of waiting for a stair at the step's end. Dividing by the step turns the count into a *rate*, trades per 24,576 s, so scores compare across pools and chains. A pool trading at its own pace settles at a decayed count of $2m=8$ whatever that pace is [derived in the code comment: an interval is $1/m$ of a step, each tick keeps $c-c/(2m)$ and adds one, the fixed point is $c=2m$ with $m=4$], which is why the rate, not the count, ranks. A silent pool reaches zero and full evictability after $32h=128\tau$: 786,432 s, about 9.1 days, on the unmeasured step; 7,680 s, about 2.1 hours, for a pool that trades every 60 s (τ = 60 s, h = 240 s); 128 days for a pool that trades daily (τ = 86,400 s, h = 345,600 s) [derived].

The clock itself is a moving mean. An anchored tick that closes an interval Δt moves τ one eighth of the way to Δt, rounded up (`TAU_EMA_SHIFT` = 3), so a steady cadence is reached instead of stalling in a dead zone; the first interval starts from $24{,}576/4=6{,}144$ s so that no single interval is read whole. A tick is *anchored* only if the smaller side of the swap moved at least 2^{-16} of the pool's mass, so dust pays gas and moves no clock. [checked in code: C:3286-3319, C:2932-2936] The constants were fixed on a replay of 11,761 real swap intervals (2026-10-27: a fast and a medium pool on Base and on Arbitrum, one long-tail pool on Arbitrum): at $m=4$, no live pool died between two of its own swaps (0 of 11,761; $m=2$ lost 3, $m=3$ lost 1), and a stopped pool died after 83 to 249 of its mean intervals. [recorded with the constants: code comment C:2915-2924]

A single swap revives a decayed pool, 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 (and a dust swap is not even anchored). The scorer keeps two zeros apart: a *dead* slot (empty, never ticked, or past the horizon) scores a true zero, drops to the back of the probe order and loses its seat first, while a *live* slot whose small rate merely rounded away is floored to one, so a real, recently touched

pool never vanishes on a rounding artefact. The window is 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.

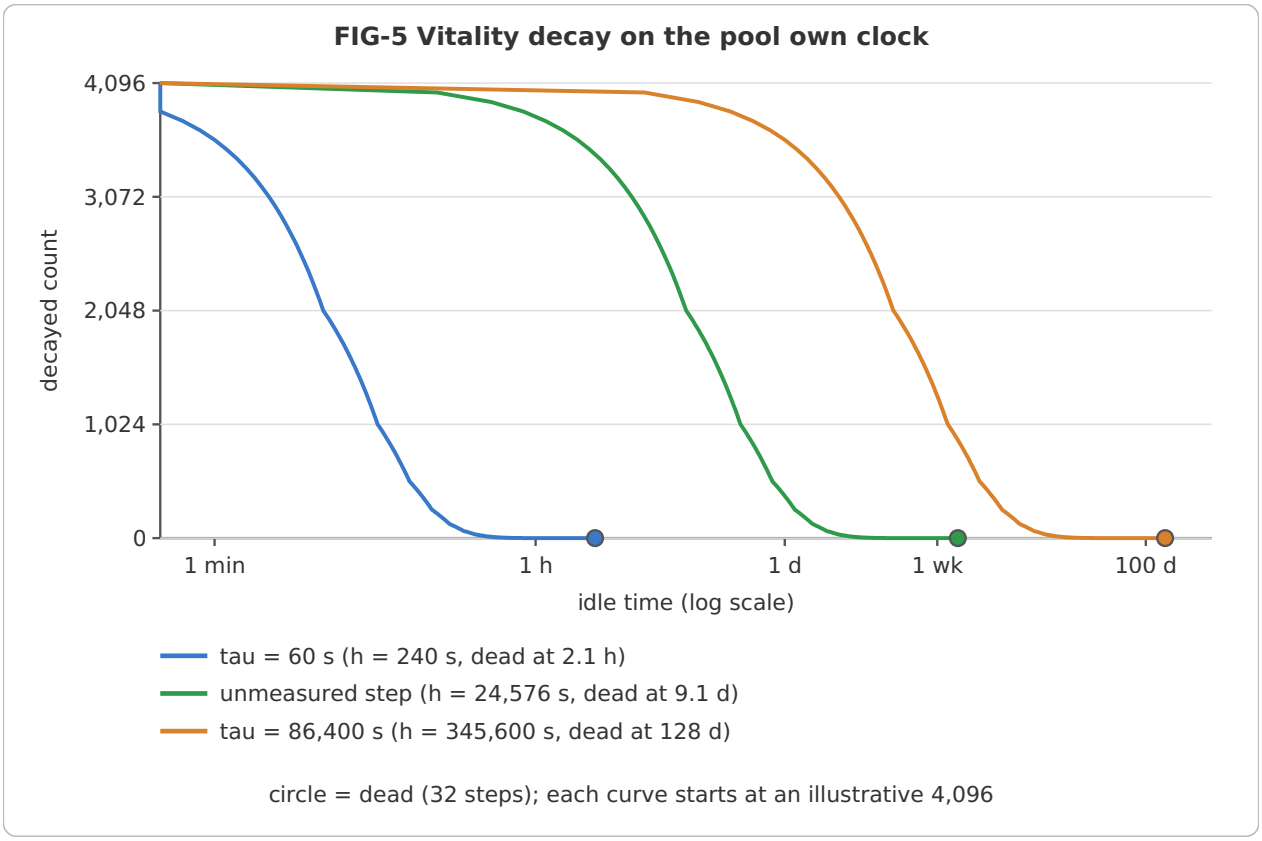

FIG. 5 Vitality decay on the pool's own clock

FIG-5 — Vitality decay on the pool's own clock. Three curves of the decayed count against idle time, one per pool: τ = 60 s (h = 240 s, dead at 2.1 h), the unmeasured step (h = 24,576 s, dead at 9.1 d), and τ = 86,400 s (h = 345,600 s, dead at 128 d); each starts at an illustrative 4,096 and halves per step with the chord inside it; a swap on the middle curve revives it from the decayed base.

TAB-9.2 — worked example, derived above: the same 32-step horizon on three pools, in the pool's own seconds. A pool with a measured τ dies after $32h$, which is 128 mean intervals.

τ	STEP h ($= 4\tau$ WHEN τ IS MEASURED)	DIES AFTER $32h$	IN MEAN INTERVALS
60 s	240 s	7,680 s (2.1 h)	128
unmeasured	24,576 s	786,432 s (9.1 d)	n/a
86,400 s	345,600 s	11,059,200 s (128 d)	128

§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`):

$$\Psi_{\text{new}} \;>\; \Psi_{\text{worst}} + \lfloor \Psi_{\text{worst}}/4 \rfloor, \qquad \Psi_{\text{new}} = 2^{\,b(d_{\text{new}})}\left(1 + \tfrac{2{,}500}{10^4}\,\mathbf{1}_{\text{bridge}}\right) \tag{E13}$$

[checked in code: H:93, H:2037-2075] The newcomer is scored conservatively: vitality one, its own depth bucket, no concentration bonus, and a zero-depth newcomer is refused outright. It now *does* carry the pair's bridge term, because the term is uniform across the pair: scored without it, the newcomer met a bar raised by a term only its rivals were given. 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, times 1.25 on a bridge pair, 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. When the seat is actually taken, the eviction ranks by $(\Psi \ll 4)\,|\,\text{bucket}$, so when every row has aged to $\Psi = 0$ the shallowest row goes, not the lowest index. [checked in code: H:2149-2170] A heavily traded incumbent is hard to displace until its decay does its work: on the unmeasured step, two idle days cost it seven halvings [derived: 172,800 / 24,576 = 7.03]; on a fast pool's own clock the same idleness is far longer in its own intervals. A permissionless V4 claim clears the same margin with the pool's measured liquidity, converted to token-denominated virtual reserves on the short side, 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. [checked in code: S:123, S:130, S:1658-1687] 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. [derived from §11 and §9.6]

§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` (0x2FF); a CREATE2 mode with a zero init-code hash; a mode bound to a kind its row of `MODE_KINDS` does not allow (a V2-salt slot to anything but V2 or Solidly, a clone slot to a non-Solidly kind, a V3-salt slot to anything but V3, Algebra or the Gate's kind 9, and kind 9 on no other line); an Algebra registration that does not declare every fee as the zero sen-

tinel; 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 live row accepts only an identical re-add, so a factory whose runtime code moved cannot be re-armed and a derive row cannot be converted into an ask row. [checked in code: H:704-832, H:877-887] `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. [checked in code: H:543-562, H:621-624] Bridges are at most `MAX_BRIDGES` = 3; `addBridge` is a curator power that survives renunciation (a duplicate add is a no-op, a fourth is `HubE(7)`), `removeBridge` a control power that dies with it. [checked in code: H:649-668]

§9.6 The structural boundary

One sentence bounds everything above: **Ψ can hide a venue; it can never misprice one.** The score orders the probe and decides eviction; it truncates nothing; no quote, no floor and no allocation reads it, because those are computed from in-transaction measurements, so memory reaches route selection only by omission. [derived from §4.4, §11 and §9.2: Ψ is an argument of the probe order and of eviction only] 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. The registry decides what is seen; it has no vote on what is paid. This is the monotonicity rule of §2.4 stated for the registry: nothing the caller can write into memory improves what the caller is paid. For a Gate wrapper the property is also pinned by test: its quote is the same at $\Psi = 0$ and at $\Psi \geq 2^{22}$ (`test_GatedAuto_GateQuoteIsFlatInPsi`).

§26 The machine underneath

Plain: this chapter lists the engineering choices that let the rules of the earlier chapters hold on a real chain, and says what each one costs. *Precise:* the invariants of Part I are only as strong as the substrate that carries them, so each choice below is stated with the check that would catch its failure.

§26.1 The size gate, measured in the build

Every deployed contract must fit EIP-170 (24,576 bytes of runtime code). The CI job "EIP-170 size guard (release profile)" runs `forge build --sizes src` with solc 0.8.36 and fails at 24,000 bytes, 576 bytes before the protocol limit; the same limit is enforced in `test/DeployedSizeGate.t.sol`. [checked in code: README:160; measured: CI job 137472871936, 6387350, 2026-11-23] At the commit this paper cites the runtime sizes (bytes, with margin to 24,576) are: Router 22,717 (1,859), Hub 21,780 (2,796), Core 19,303 (5,273), Solver 21,349 (3,227), Quoter 19,892 (4,684), Gate 10,758 (13,818), Math 9,768 (14,808), GateWrapper 4,414 (20,162); the eight adapters range from 3,014 to 6,347. [measured: equal to the README table and to `.sizes-snapshot`] The largest initcode is the Solver's, 31,508 bytes, under the EIP-3860 limit of 49,152. [measured] The margins are small on purpose to see: the Router has 1,859 bytes left, so every guard added is paid for in bytes, which is why §2.7's rule "each guard with its mutant" is also a size budget. [derived: margin column] *TAB-26.1 — runtime sizes at the cited commit, in bytes, with the margin to EIP-170 (24,576).*

CONTRACT	RUNTIME (BYTES)	MARGIN TO 24,576
Router	22,717	1,859
Hub	21,780	2,796
Core	19,303	5,273
Solver	21,349	3,227
Quoter	19,892	4,684
Gate	10,758	13,818
Math	9,768	14,808
GateWrapper	4,414	20,162

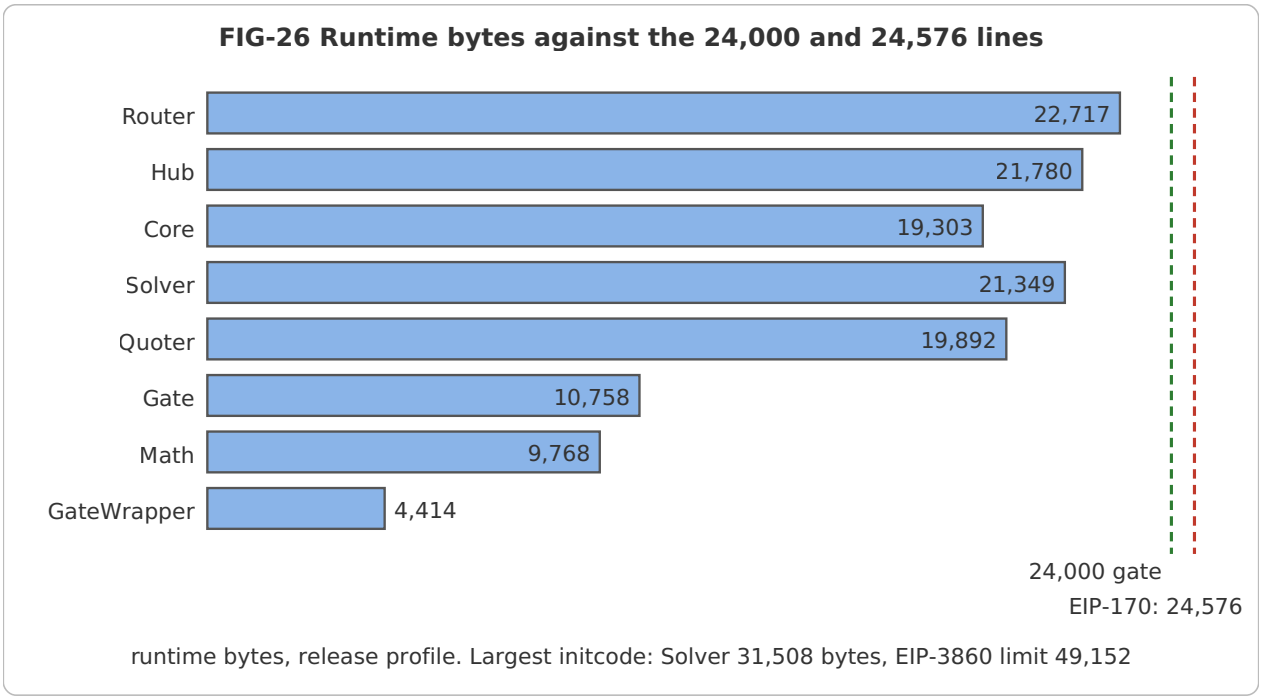

FIG. 26 Runtime bytes against the 24,000 and 24,576 lines

FIG-26 — bar chart of runtime bytes against the 24,000 and 24,576 lines.

§26.2 The transient lock

The Router keeps its re-entrancy lock, and nine other per-swap slots, in transient storage (`tload` / `tstore`), which the EVM clears at the end of the transaction: a lock that cannot outlive the call that set it. The modifier `nrEntrant` reads the slot `keccak256("blaze.r.lock")`, reverts `RouterE(7)` if it is non-zero, sets it to 1, runs the body and sets it to 0. [checked in code: Router:193, 307-314] The scope is one lock per top-level entry, not one per hop. Five entry points carry it: `swapExactIn`, `swapExactInWithPermit2`, `swapExactInNative`, `swapBestExactIn` and `claimFee`. [checked in code: Router:347, 444, 465, 518, 558] The one internal bridge, `selfExecutePrePulled`, is deliberately not guarded: it is callable only by the Router itself

(`RouterE(1)`) and the outer entry already holds the lock, so a guarded inner call would always trip `RouterE(7)`. [checked in code: Router:597-606] A revert rolls the lock back with the rest of the transient writes (EIP-1153 semantics, as the comment at Router:2060 relies on), so a failed swap cannot leave the Router locked.

What the lock costs is derivable from the EVM, not measured here: `TLOAD` and `TSTORE` cost 100 gas each (EIP-1153), so the lock is one load and two stores, 300 gas of opcodes; a persistent-storage lock would pay 20,000 for the first `SSTORE` from zero plus 2,100 for the cold slot, 22,100 before any refund (EIP-2200, EIP-2929). [derived: EIP constants; no test in the repo isolates the lock's gas, the gas instrument measures whole transactions, `isolate = true` in foundry.toml] The refusal is witnessed from the side that is easy to forget: `ReentryFromTheOutputTransfer` re-enters from the output token's `transfer`, mid-swap, and asserts the nested call is refused by `RouterE(7)` itself and not by a later guard. [checked in code: test/ReentryFromTheOutputTransfer.t.sol header] The README lists "transient lock" among the public guarantees; this section adds no guarantee beyond the code lines above.

§26.3 The callback is authenticated by a committed pool

A V3-shaped venue (V3, Algebra, and a Gate wrapper, which speaks the same swap and callback ABI) asks to be paid from inside its own `swap`. The Router answers in a `fallback()` that accepts any selector, so long as the calldata holds at least 4 + 64 bytes, and hands the two words to `_v3Callback`. [checked in code: Router:2107-2117] The authentication is a committed address, not a derived one. Just before it calls `pool.swap`, `_execV3Amt` writes three transient values: the pool of this leg, the token it will be paid in, and the most it may be paid, which is the leg's input. [checked in code: Router:1998-2003] The callback then requires `msg.sender == expected` and `expected ≠ 0`, else `RouterE(6)`; it takes the amount owed from the callback, refuses zero and anything above the committed maximum with `RouterE(8)`; and it clears all three values before it pays, so a pool that re-enters the callback reads `expected == 0` the second time and reverts `RouterE(6)`. [checked in code: Router:2146-2165] After `pool.swap` returns, the three slots are cleared again, and a pool slot that is still set means no callback ran: the leg's input is then counted as zero, never as paid. [checked in code: Router:2010-2019] The Router also measures that the pool received what the callback paid (`RouterE(13)` for a fee-on-transfer input). [checked in code: Router:2172-2174]

CREATE2 is therefore not part of this check. The derivation of §7.1 (`create2Address`, `deriveAddress`) lives where a pool is discovered and attested, in the Hub and the Core, and the Router never calls either of them. [derived: `grep -n 'create2Address\|deriveAddress'` on the Router at 6387350 returns nothing] What bounds a fake pool in the Router is the committed maximum (a callback can never take more than this leg's input), the single-shot clear, the measured delta, and the per-leg floor, which requires the leg to deliver at least `BPC.LEG_FLOOR_BPS` of its bound (80 %, rounded up) whenever the leg has a bound (an attestation or an in-frame quote), or the whole swap reverts `RouterE(5)`. [checked in code: Router:1748-1749, 1894; Core:405] Three tests keep the guard honest: `CallbackAuthImpostorDrain` (a swap is in flight, a different contract asks to be paid, the assertion is the impostor's balance), `V3ContextDiesWithTheSwap` (the context is cleared when a pool returns without calling back) and `FallbackCalldataLengthGate`. [checked in code: the three files' headers at 6387350] The mutant that each kills is stated in those headers for the first two (deleting the sender check; deleting the clear after the swap).

§26.4 Assembly where it is measured

The contracts are Solidity 0.8.36 with inline Yul, built with `via_ir = true`, `optimizer_runs = 200` and `evm_version = "cancun"` (the version that has `tload`, `tstore` and `mcopy`) in both the default and the release profile, with `bytecode_hash = "none"`. [checked in code: foundry.toml:6-20, 53-77] There are 96 inline-assembly blocks in the seven top-level source files of `src/` (eight contracts: the GateWrapper lives in Gate.sol), by contract Core 37, Router 33, Solver 12, Quoter 7, Hub 4, Math 3, Gate 0, GateWrapper 0. [measured: `grep` for an `assembly` opener outside comments, 969bcc6] The comment in `foundry.toml` says why `via_ir` stays on: several functions (`Solver._buildHop`, `Math._weights`, `Router._execute`) are written assuming the stack-slot folding of that pipeline, and without it the build would likely hit "stack too deep". [checked in code: foundry.toml:10-12]

Three uses, each with the reason the source gives. (1) Transient context: the lock and the callback's committed pool are `tload` / `tstore` blocks written by hand (§26.2, §26.3). (2) Exact gas arithmetic: `Solver._walkOnItsOwnBudget` rebuilds a plan as a static self-call and sizes its gas by hand, W = min((SOLVE_B - s - 20,000) * 63 / 64, WALK_CAP), so that EIP-150's 63/64 rule hands the call the whole W; it splices the answer back without ABI decoding. [checked in code: Solver:317-353; the threshold is tested in test/CapacityWalkBudget.t.sol, named in the Solver comment] (3) Memory moves: `mcopy` for the candidate arrays in the Solver and the Math. [checked in code: Solver:335, 348; Math:128]

The one gas figure the source itself gives for a choice of this kind is not an assembly figure: merging the two reads `token0()` and `token1()` of a leg's pool into one snapshot saves 1,498 gas for one leg up to 7,355 for five (1.65 % of that transaction) and 58 bytes of Router code, and closes a crack, since a pool that answered the two reads differently could fabricate `legOut == tokenIn` and switch the leg floor off. [as stated in the code comment, Router:1905-1915] The G7/G8 gas campaign (`docs/GAS_CAMPAIGN_G7_G8.md`) measures whole routes per chain, venue kind and discovery mode (quote, record, execute); no cell in it isolates an assembly block, so this paper gives no gas figure for Yul as such. [checked in code: that document's section 1 and 4]

§26.5 A stateless library, pinned by hash

The per-hop choice lives in `BlazePhoenixMath`, a library with no storage of its own (no `sload`, `sstore` or `storage` pointer in it), called by the Solver. [measured: grep of the Math source at 969bcc6, 0 hits] The Solver reaches it by `delegatecall`, one per hop: `select` is a `public view` function of the library, and the Solver's own comment says "ONE delegatecall per hop, static inside the solve's staticcall (EIP-214)". [checked in code: Math:20-23 (header: the Solver delegatecalls it once per hop; called from a STATICCALL the delegatecall stays static, EIP-214), Math:120-123, Solver:790-795] The library works on its own ABI copy of the arrays and answers with a permutation `map`, which the Solver replays on its own arrays. The Solver then checks that the returndata length is exactly the canonical ABI encoding of the values it decoded, and reverts `SolverE(7)` otherwise, because the ABI decoder alone would accept a longer returndata. [checked in code: Solver:812-816]

The Solver refuses to deploy if the library's deployed code differs from the code it was built with: the constructor compares `address(BlazePhoenixMath).codehash` with `keccak256(type(BlazePhoenixMath).runtimeCode)` and reverts `SolverE(6)` on a mismatch. [checked in code: Solver:200-206] The comment gives the reason the pin is sufficient: a library with no state-changing function carries no deploy-address guard and links no other library (the Core it uses is all internal), so its deployed

code is exactly `runtimeCode`. The linker puts the library's address as a `PUSH20` at every call site, so there is no storage slot and no setter by which it could change. [checked in code: Solver:200-205] The Gate and the adapters declare no `VERSION`.

Part III — Routing and execution

§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.

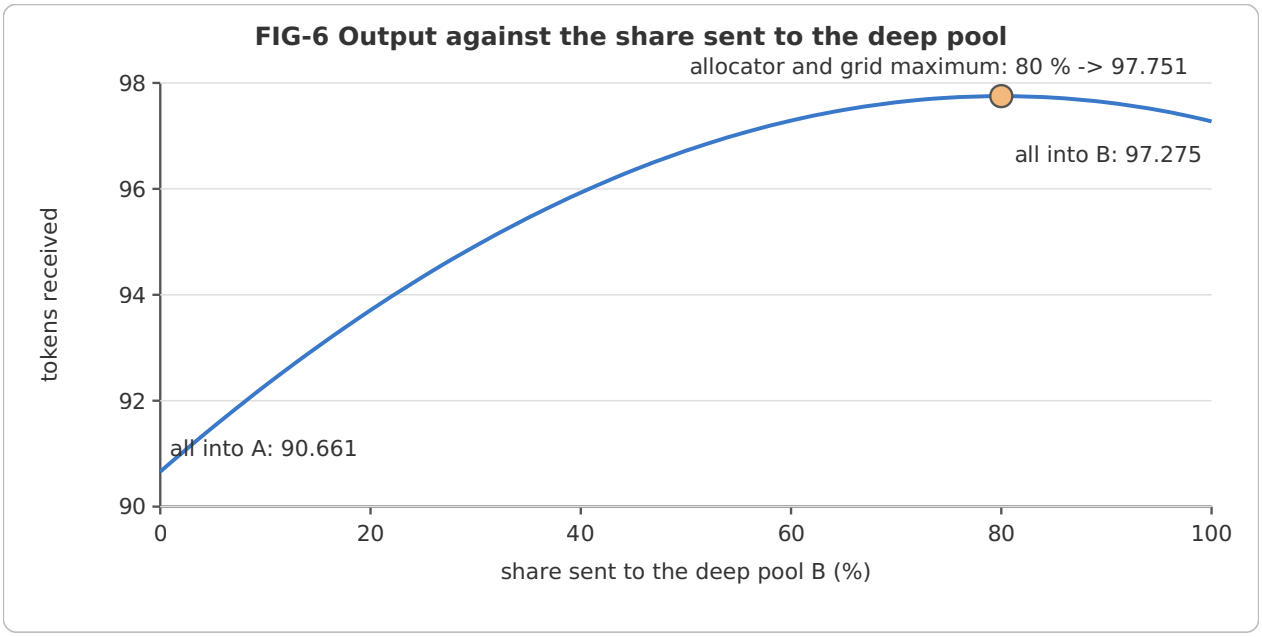

FIG. 6 Output against the share sent to the deep pool

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 four topologies and returns the maximiser (`findBestRoutePlan` , with the runner-up as a fallback): a *direct* route in one hop and, for each configured bridge, a *bridge* route through that one intermediate (`tIn → bridge → tOut`). Two hops is the deepest shape the Solver builds and the most the Router accepts (`MAX_HOPS = 2` ; a longer route is refused with `RouterE(3)` at every door). The topologies are judged in order and a tie keeps the earlier one, but the bridge topologies only add candidates to the judgement: a direct route still wins whenever it delivers more, because the comparison is by the output built and not by the depth of the topology. Bridges come from an operator-configured set of at most three tokens in the Hub (`BRIDGE_SLOTS = 3` , the Hub's `MAX_BRIDGES`), so `TOPOLOGIES = 1 + BRIDGE_SLOTS = 4` ; the Solver expands every configured bridge and lets the measured `totalOut` decide, per trade. Every hop splits across parallel legs under one budget with one producer: `MAX_LEGS_PER_HOP = 5` in Core is the Router's ceiling (`RouterE(3)` refuses a wider hop), the Quoter's test for whether a plan is executable, and the Solver's leg budget for every stage of every topology, so the Solver never proposes a hop the Router refuses and never stops a leg short of one the Router takes. A route therefore holds at most `MAX_HOPS × MAX_LEGS_PER_HOP = 10` legs, and that bound is the product of those two constants and nothing else: no code needs a route-wide count, so no third constant can drift from it. 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.

§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); and orders the union by Ψ. No count cuts the funnel: Ψ

fixes only which venue is probed first and settles exact ties (an exact tie breaks toward the lower fee, so on a cold registry the order is a function of price rather than of history), so a deep pool listed behind several thin venues is seen, weighted and can displace them instead of being starved by list order, and the quotes make every real decision. The cut to `MAX_LEGS_PER_HOP` = 5 legs in a hop comes after the probe and the weighting, and the candidates that pass the believability band reach the split gate, not only those the cut keeps (§3). What leaves the list is dust, and dust is measured against the pair rather than counted out. Each row is weighed by its mass on this hop, the output token it can pay (`hopMass18`), and row j is dropped if and only if $d_j \cdot 10^6 < \text{MIN_SPLIT_IMPROVEMENT_PPM} \cdot (D - d_j)$, with D the sum of the masses. The rule has no threshold of its own and no price: it is the split gate's improvement bar (§10.4) read as a share of the pair's measured depth. For pools at one price and one fee, the most a leg on j can add to what the others pay, relative to it, is $d_j/(D - d_j)$ at every size (the derivation is in the `_dropDust` documentation of the Solver), so a dropped row cannot lift an order by the improvement a split must show to be kept. A row alone on its pair or holding all of it is never dust, nor is the deepest row, and a Gate wrapper, whose mass is not read here, stays: absent is not zero. A singleton book (V4) is measured twice: its declared side is its current tick alone, not a bound, so before a book that declares mass is dropped every book is walked over its tick and the next stretch (`FUNNEL_STRETCHES = 2`): its own d_j is the larger of the side it declares and what it holds over that walk, and what it adds to the other rows' D is what it holds over it. The limit is stated where it is decided: at two prices inside the band a dust row priced better can win an order small enough for it, and dropping it costs at most the price gap.

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' prices** (`_depthWeightedMedian`): prices are sorted with their depths travelling alongside, and the median is the price 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. The band reads **prices, not rates**. A probe measures the rate after the venue's fee, so a band on rates reads a fee as a price: a venue charging 10 % at the honest price sits 10 % under an honest 0.3 % venue. A probe's price is $p_i = r_i \cdot 10^6/(10^6 - f_i)$, with f_i the fee the probe itself charged, in pips.

The width is no-arbitrage, not a number of basis points. Two honest venues at rest leave no round trip to take, so their spots satisfy $(1-f_i)(1-f_*) \le s_i/s_* \le 1/((1-f_i)(1-f_*))$, with $*$ the base; a measured price sits within its probe's error bound e of its spot, $s(1-e) \le p \le s/(1-e)$. With $k_i = (1-f_i)(1-f_*)(1-e_i)(1-e_*)$, every honest venue at rest has $k_i \le p_i/p_* \le 1/k_i$. The band keeps row i if and only if

- below: $p_i \ge p_*\, k_i$, with e_* read at the hop's size;
- above: $p_i\, k_i\, (1-f_*) \le p_*$, the base's fee counted twice.

Both hold for every honest venue at rest, so the band never drops one. The base's fee counts twice above because k_i alone lets a base holding half the mass park itself on the no-arbitrage edge, where it leaves a round trip of only about $e_h + e_*$ open — too little to pay an arbitrageur's gas — drops the honest venue h over the top and charges the user $1-(1-f_*)^2$ against h (19 % at $f_* = 10$ %). Counted twice, dropping h needs a round trip G between h and the base with $1+G > 1/(1-f_*)$; then the user's loss against h, $1-(1-f_*)^2/(1+G)$, stays under $1-(1+G)^{-3} \le 3G$ (the first step from $1-f_* > 1/(1+G)$, the second is Bernoulli's inequality): hurting the user by L leaves at least $L/3$ open to whoever closes it. Below, k_i alone loses nothing: a row dropped there has a measured rate under the base's own. A fee nobody knows (a Gate wrapper reports none) is handled without inventing one: its price is read as its rate, a floor of the truth; above it is judged with $f_i = 0$; below it is never judged; its mass does not vote while a row with a stated fee does; and a base whose fee is unknown states no width, so every live row is kept. A candidate set of one is not filtered at all, and a set in which no live row carries a vote (every live depth zero, or every vote muted) falls back to the plain median's row, which is live, so an otherwise routable hop is not killed for want of a vote.

The width has a price, and it is stated where it is decided. The band is as narrow as the fees: where the venues charge 0.05 %, an honest venue that lags the base by more than the round trip is dropped until an arbitrageur closes the gap. A venue at a fair price behind a large fee is now inside the band, and the split weighs by depth, so the split carries a rule for it (§10.4, *every leg pays for its place*). A fee near 100 % leaves k_i near zero and passes both edges, but such a venue's rate is near zero and the gate weighs rates. What the band does not bound is the arbitrageur's gas and the quote shapes the probe lists as its limits. The metamorphic lane of §15.7 measures the band from outside, with one pool priced 10^{10} away from its sibling : before the price band existed, the header of that test file records that the plan paid 35 % less than the outlier alone (MR-R1); the relations now run inside the band's domain, so that figure is history, not a property asserted at 6387350. The source of this subsection is the documentation of `BlazePhoenixMath` («THE BAND») at 6387350.

§10.4 The allocator and its four 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. Four 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 caps the promise and, on the first hop, cuts the leg's *committed input* in the same ratio (a later hop keeps its input, which the Router rescales to what the stage before produced), 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, the capital stays, because cutting it there was measured to force partial fills on pools that execute fine: every topology ranks the leg at the cap, and on the winning topology's own build it is promised what the pool's book pays for that input, walked read-only and held between the cap and the quote (a book that cannot be read keeps the cap). 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 three values that had never been calibrated against a measurement. When the gate collapses a route, it measures every survivor of the band alone at full size, not two representatives: the deepest pool and the best marginal rate are both heuristics for the best single leg, and with three or more survivors that leg can be neither (`test/SplitGateSeesEverySurvivor.t.sol`); the best fill wins.
- **Every leg pays for its place.** The weights are depths, and a split by depth is the best split only where every leg pays one price. Because the band judges prices (§10.3), a venue at a fair price behind a high fee sits inside it and takes its depth's share of the order: in the test that pins this rule, beside two honest venues at 0.3 %, a venue at 10 % holding 0.3× their depth took 23.1 % of an order a tenth of each, and the route paid 1.10 % less than without it (`test/SplitLegPaysForItsPlace.t.sol` ; figures measured at 3975a50 and recorded in its header). The gate above asks whether the whole split beats its best leg alone; this guard asks it of each leg. A leg of a split of three or more stays only while the split with it pays at least `MIN_SPLIT_IMPROVEMENT_PPM` more than the same split without it, and a split of two is not tried, since without one of its legs it is the other alone, which the gate measures at full size. No threshold of its own decides *which* leg is tried: an optimal split leaves leg j out if and only if its rate at

zero is at most the common marginal rate m of the legs it keeps; some kept leg i receives at least its present share a_i there, and on a concave book $m \le \text{marginal}_i(a_i) \le \text{average}_i(a_i)$; a probe pays its own impact, so j's probe rate is at most its rate at zero. So j can be left out only if its probe rate is at most what another leg pays on average now, and only then is a leg tried, the one whose probe paid least first; the trials stop at the first leg that pays. The leg left out frees its weight to the others and stays in the band, so the gate still quotes it alone. The cost is one more quote of the legs that remain per trial.

- **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
`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

DOOR	SIGNATURE (ABRIDGED)	AUTHORISATION	ROUTE
`swapBestExactIn`	`(tokenIn, tokenOut, amountIn, userMinOut, recipient, deadline)`	classic	solved in-frame by the Solver (`findBestRoutePlan` , under the `SOLVE_GAS` budget of 14,700,000), 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. If the in-frame Solver reverts without data, the door refuses with `RouterE(17)` rather than guessing. An EIP-7702 delegated account calls the Router as a contract would; no separate entry exists for it, and the executor itself refuses to run inside a delegated frame, where every balance it measures would be the account's (`RouterE(18)`).

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 two hops (`MAX_HOPS = 2`), more than five legs in a hop (`MAX_LEGS_PER_HOP = 5`), a hop whose two tokens are the same, 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. The product of the two ceilings, $2 \times 5 = 10$ legs, is the most a route can hold; the Solver's own budget is set to the same shape (§10.1).

§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); the second 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 Four refusals 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, joined by the quote measured in-frame as the second element of a maximum: $\text{bound} = \max(\text{attested}, \lfloor q \cdot 5{,}000/10^4 \rfloor)$, where q is the in-frame quote and `MIN_QUOTE_COVERAGE_BPS = 5,000` (Router:1789-1842 at 6387350). 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. And it is monotone in the attestation — attesting more never lowers the floor — because the lift goes to the threshold itself and not to the whole measurement. The guard runs whenever there is any floor basis — the caller's attestation or the in-frame quote — so writing zero into a route does not switch the caller's own floor off. A venue with no measurable quote and no attestation is bounded by the aggregate floors and the user minimum.

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. Only legs that have a bound enter the sum, on both sides, so a leg without one cannot mask another's shortfall. For one leg the rule collapses exactly onto the per-leg floor — zero new rigidity, by construction.

The declaration cap. A leg's measured output delta may not exceed what the venue declares it paid (`RouterE(20)`, Router:1787 at 6387350); less is fine, which is how a fee-on-transfer output token passes. The window of a leg also spans the venue's own code, and tokens landing on the Router inside it must not be counted as the leg's output.

The gated door. A leg of an external kind executes only if the wrapper it names is the one the Hub's Gate derives from the hop's own tokens and fee (`RouterE(19)`); the kind in calldata is a claim, not an identity (§14.4).

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 re-

fuses 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 derives the pool id from the leg's own fields and refuses a leg whose id or direction does not match (`RouterE(11)`), then 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`, `RouterE(3)`), 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. It is an encoding rule, like token ordering: every route can be written in the canonical order, so nothing is rejected that could not be re-expressed.

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`, which holds the compounded measured net ratio in WAD), 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); each per-leg bound on a constant-product leg is re-priced by that leg's own measured ratio (what the pool received over what was sent to it, rounded up); 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`):

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

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

Three constants decide it: `FLOOR_BASE_BPS = 9,600`, `FLOOR_PER_LEG_BPS = 200`, `FLOOR_HARD_MAX_LOSS_BPS = 2,000` (Core:400-402; the floor itself is Core:3345-3357, the leg shave Core:3338). 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 $[\text{attested},\ \text{attested}/(1-\text{fee})]$, the attested floor read back on the gross output (one factor per charging hop); with the fee off the output, it equals the attested floor.

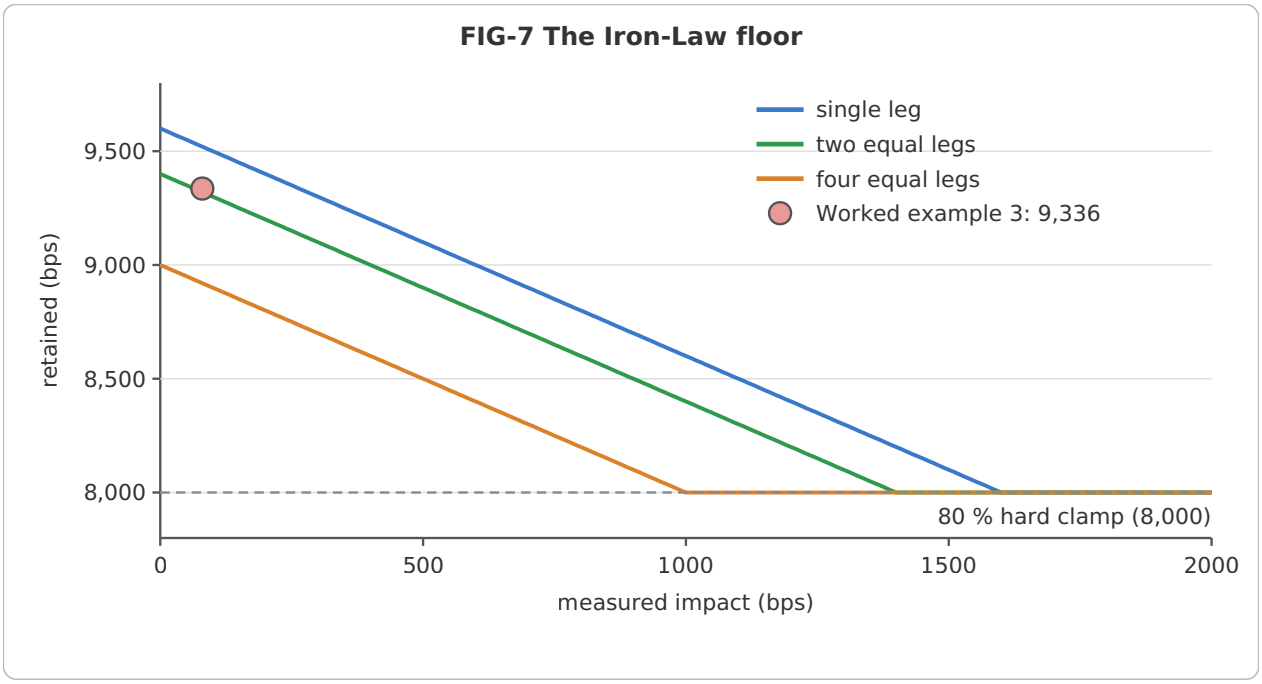

FIG. 7 The Iron-Law floor

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 72 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 delivery 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 (`RouterE(5)`) — 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 `Swap(payer, tokenIn, tokenOut, amountIn, delivered, declaredLegs)`, with `amountIn` the gross amount the user handed over, and `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, published net of any transfer tax the output token levied, on the transfer into the Router and on the final transfer (scaled by received over the venues' declared gross and by delivered over sent; the identity on an untaxed token), so a taxed output never shows a floor as met that the tax already took. 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 caller — a control call, a treasury move by a non-treasury, a fee claim with nothing owed, the self-call door called from outside	`onlyControl`, `moveTreasury`, `claimFee`, `selfExecutePrePulled`
`RouterE(2)`	paused, or renunciation attempted while paused	`whenLive`, `renounceControl`

CODE	MEANING	WHERE
`RouterE(3)`	bad input — zero address, unwired native door, hop count (over 2), legs per hop (over 5), hop discontinuity, a hookless leg after a hooked one	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, or a hook that runs in the swap is paused in the Hub	V4 leg pre-commitment
`RouterE(10)`	`userMinOut == 0` with `amountIn > 0`	every door
`RouterE(11)`	V4 leg whose pool id or direction does not derive from its own fields	V4 leg pre-commitment
`RouterE(13)`	fee-on-transfer input on a V3 leg — the pool measurably received less than it demanded	`_v3Callback`
`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`
`RouterE(17)`	the in-frame Solver reverted without data	`swapBestExactIn`
`RouterE(18)`	executor entered inside an EIP-7702 delegated frame	`_execute`
`RouterE(19)`	external-kind leg whose Gate wrapper does not derive from the hop's tokens and fee	`_execScaled`
`RouterE(20)`	a leg's measured output exceeds what its venue declares it paid	`_execScaled`

Code 12 is 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.

§11.8 Fee-on-transfer on the way out

§11.4 treated the input side: a token that taxes its own transfers enters the route at its measured size. The output side has two transfers, and the Router measures each. The first is the venue paying the Router. The Router's balance delta over its entry baseline is the gross output of §11.6, and the venue's own declaration of

what it paid bounds that delta from above (`RouterE(20)` , §11.3). The second is the Router paying the recipient: the recipient's balance delta is the delivered amount, and `userMinOut` is enforced on it (`RouterE(5)`). Both are balance deltas; neither trusts a nominal amount.

What was left to settle is which of the two the *protocol* floors read. Up to 969bcc6 the route floors of §11.5, the protocol floor and the caller's `singleOutFloor` , were compared with the measured delta, which is net of any tax on the first transfer, while both floors are priced on gross output. At 6387350 (PR #132) they are read instead against `outCap` (Router:1594-1600): the gross the venues declared paying in the output token, net of what the route took back out (Router:1337-1338 accumulates it, and it also caps the measured delta, Router: 1490). The user's bound does not move: `userMinOut` stays on the recipient's delta. The per-leg floor and the hop budget keep reading the measured delta, with their base taken from the attestation.

Three consequences follow from the definitions, not from a measurement, and are stated as such. First, on an untaxed token the venue's declaration equals the measured delta (§11.3, declaration cap), so `outCap` and the measured amount are the same number and an untaxed swap settles as before. Second, whatever the token does, the user receives at least `userMinOut` or the swap is refused: the check is a refusal on the recipient's measured delta, so delivered $\geq$ `userMinOut` on every settlement. The protocol floors are conditions on the route's pricing and are tested before the second transfer (§11.5); this change does not make them depend on what the second transfer does. Third, a tax large enough to take a leg below its 80 % bound or a hop past its budget is still refused (`RouterE(5)`), because those two guards still read the measured delta. What the change removes is one false refusal: a route that the venues priced correctly, rejected by a floor because the output token taxed the venue's payment.

§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 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 fallback route and a flag	the Solver's plan, packed with the fee and the safety buffer (`BlazePhoenixQuoter.sol:231`); a `view`
`previewPlanWithMinOut(…, userMinOut)`	the same	`canExecute` judged against the caller's minimum, which can only tighten the floor (`:243`); a `view`
`previewRoute(route, userMinOut)`	a `Preview`	a route built earlier or off-chain, packed without re-running the Solver (`:316`); a `view`
`batchQuote(entries[])`	up to `MAX_BATCH = 32` previews	a pair that cannot be planned leaves a zero-initialised entry; more than 32 entries revert `QuoterE(4)` (`:173` , `:323-327`); a `view`

FUNCTION	RETURNS	NOTES
`previewAndEncode(…)` / `previewAndEncodeWithMinOut(…)`	a `Preview` and the calldata for `swapExactIn`	the way an integrator takes a quote; the §12.2 study uses exactly this. Not a `view`: call it by `eth_call` (`:264`, `:274`)
`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; by `eth_call` (`:811`)
`previewRouteExact(route, amountIn)`	the same, for a route built off-chain	the only way to quote a Gate leg (kind 9), which the Solver never plans; by `eth_call` (`:847`)

What a preview is. The `Preview` struct (`:204`) is advisory by design: the binding checks are the ones the Router re-derives in-frame. Its fields: `grossOut` echoes the plan's total; `protocolFee` is the effect of the 28 bps fee on the output (`Core:426`), modelled per regime exactly as the Router charges it (§13) (once on an anchored route, once per hop on an exhausted one) and rounded up as the Router rounds (`:346-470`); `safetyBuffer` is the inter-leg buffer

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

where n is the leg count and a the number of constant-product legs priced on an *assumed* fee (a declared fee the Core replaced under its ceiling). The terms are `BASE_SAFETY_BPS = 0`, `PER_LEG_SAFETY = 1`, `ASSUMED_FEE_SAFETY = 5` and the shared cap `SAFETY_CAP_BPS = 10` (`:146-148`, `:172`): zero at two legs or fewer, one basis point per leg above two, five per assumed-fee leg, under one cap of ten, because a buffer without a ceiling stops being a buffer and becomes a floor. The code's own reason for 5: only one direction hurts. A pool that charges less than the assumed 30 bps under-quotes harmlessly; one that charges more over-quotes and execution then delivers under the floor. The wired V2 venues publish 25 to 30 bps, so five covers the realistic spread, and the buffer deliberately does not cover the worst case the V2 fee ceiling admits, because a buffer sized for that case is a floor, and the floor already exists (`:150-172`). `netOut` is grossOut − protocolFee − s; `ironFloor` echoes the attested floor; `topology` (0 direct, 1 through one bridge) and `bridgeUsed` are derived from the hop count and from the fee anchor the Router will use, asked of the same producer (`BPC.feeCalendar`, reached through `feeCalendarOf`, `:346-470`), so the preview and the executor cannot name different bridges. `canExecute` carries the Router's shape and hook refusals (`_hookRefusedIn`, `:499`): an empty route, more than `MAX_HOPS` hops, a hop with no leg or more than `MAX_LEGS_PER_HOP`, a leg off its hop's pair, a hook that alters deltas or runs in the swap while the Hub holds it paused. 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 encode doors ask the Router's own questions. `previewAndEncode` returns the calldata with `effectiveMinOut = max(userMinOut, ironFloor)` already inside, and returns it empty when the route cannot settle: a caller cannot submit a transaction the function already knows will be refused. Three refusals are asked before the bytes are built (`_encode`, `:298-312`). The deadline is behind the block (`RouterE(4)`). The route, funded as the Router funds it, delivers under the floor the bytes carry, a figure taken from the venues' dry-run on every fee calendar, together with each leg's floor, the hop budget and the Router's route protocol floor (the last hop's gross output against `BPC.routeFloorBps` of that hop's base) (`_settles`, `:988`). And a concentrated leg is paid in a token its pool receives short of (`RouterE(13)`), measured by the input probe (`_inputTaxed`, `:684`): the pool is run backwards on the leg's output so that it pays this frame the input

token before it calls back, and the callback pays it back and measures. The view doors cannot make this last measurement, since a view frame holds no token; the dry-run doors can, which is why the doors that return bytes are not views.

One item's gas does not starve the next. In `batchQuote` each entry's solve is given its share of what is left, `gasleft() / (n - i)` for entry i of n (`:331-337`): a solve that runs out hands back one sixty-fourth of its gas (EIP-150), and the entries after it would otherwise read zero for a book that is not theirs. The share is taken over the entries still to run, so it never shrinks down the batch and the last entry is given everything.

The Exact Pass. For a pool-exact preview of a concentrated venue, the Quoter calls the pool's real swap entry with a price limit at the extreme; the pool calls back demanding payment; the Quoter's callback, instead of paying, reverts, carrying the computed deltas in a tagged payload (`QuoteDeltas` , so a venue's own revert data can never be read as a quote, whatever its length) (`_simConc` , `:660-675`). 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 (`unlockCallback` , `:724-742` ; `_simV4` , `:747`). The scalar the call returns is **net**, and is taken where the Router takes the fee: on the anchor hop's input on a multi-hop route (inside the walk, so the next pool rounds on the reduced amount), on the output only where the fee calendar puts it there, and rounded up, so a minimum derived from it is honoured by the floor (`_exact` , `:853-950` ; `test/QuoterExactNetOut.t.sol`). The scalar is also the smaller of the walk and the figure the Router funds, so the two producers cannot drift apart.

It answers zero, and says nothing more, in the cases where the Router will not fill: a leg the Router refuses (a delta-altering or paused hook `RouterE(9)` , a key that does not derive `RouterE(11)` , a native side with no canonical WETH `RouterE(8)`), a taxed input into a concentrated leg (`RouterE(13)`), and a pool named twice in one hop. The last case: the Router runs the legs in turn and the second meets the pool the first moved, so a split priced side by side over-promises. The Solver merges a pool's legs and never plans it; a route built off-chain can. The route keeps its pool-math attestation, so what the Router compares against is unchanged (`:933-950` , `:955`).

A pool that refuses the dry run keeps the plan's own claim, scaled and never raised. Write the venue's output as a function f of its input with $f(0) = 0$ and the plan's one known point $(b, f(b))$ with b the leg's planned input (`_priceLeg` , `:1174-1185`). For a constant-product pool with reserves x, y and retention γ,

$$f(a) \;=\; \frac{y\gamma a}{x + \gamma a}, \qquad f'(a) = \frac{y\gamma x}{(x+\gamma a)^2}, \qquad f''(a) = -\frac{2\,y\gamma^2 x}{(x+\gamma a)^3} < 0.$$

A concentrated pool is one such curve on shifted reserves inside a tick range, and at a range boundary the marginal rate γ times the price is continuous and only falls as the price moves against the trade, so f is concave there too (derived; the code's comment states the same). For a concave f with $f(0) = 0$, $f(\lambda b) \geq \lambda f(b)$ for $0 \leq \lambda \leq 1$ (the curve lies above its chord), and $f(t)/t$ is non-increasing, so for $t > b$ the same rescale $tf(b)/b$ is an upper bound that overstates. The code therefore rescales linearly below b, where the result is the chord and understates, and caps the ratio at one above it (`mulDiv(expectedOut, min(legIn, base), base)`), so a refused leg may keep the plan's claim and never exceed it. Hop 0 is funded at what the committed legs sum to, so a capacity-clamped route is not dry-run at a size the Router will never spend. 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 route's first pool, the shallow one, 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`, `N = 240`). A sample for which the encode door returns no bytes is counted as "no quote", and the test requires fewer than a tenth of the samples to be so. The tables below were measured in CI at 969bcc6: of the 240 samples, none was without a quote.

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
over 100 bps (up to 900)	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

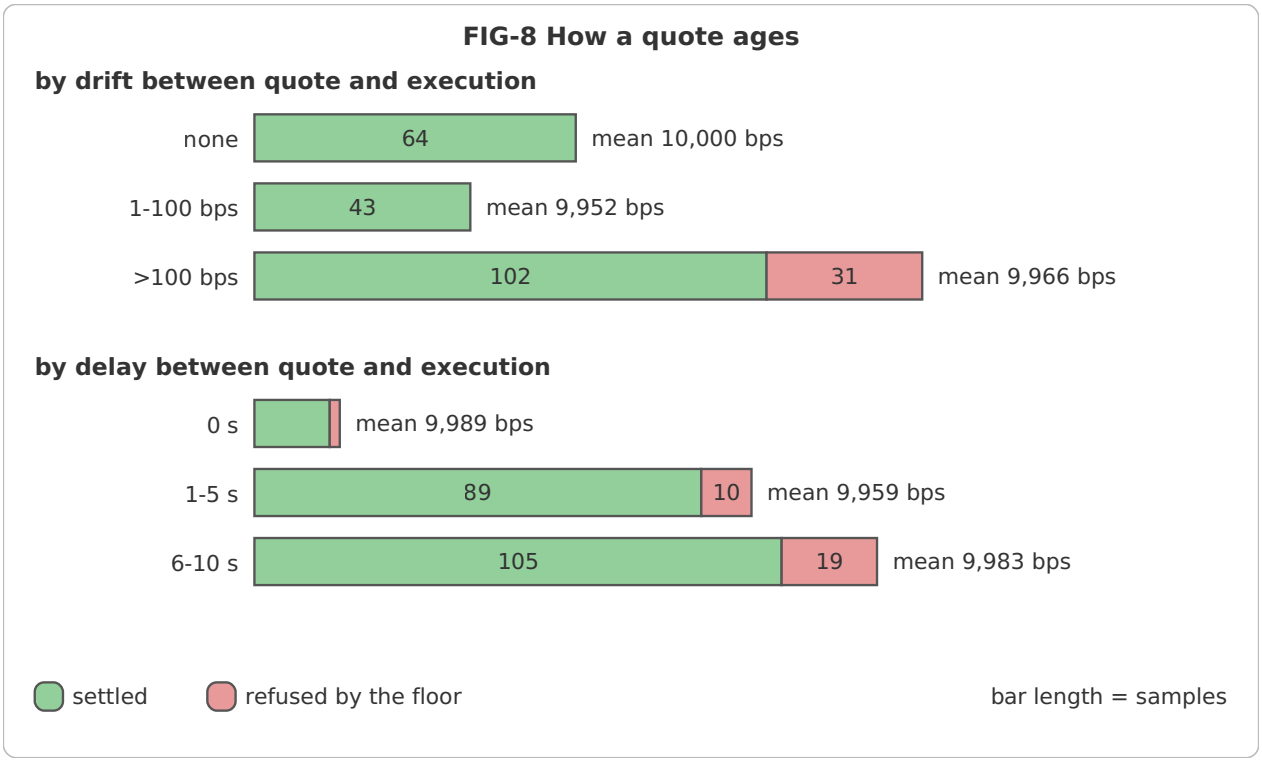

FIG. 8 How a quote ages

FIG-8 — How a quote ages: settled and refused samples by drift bucket, with the mean delivered/predicted ratio of each bucket's settled samples; 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 at least its own net 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 (`test_QuoteExecutedAfterItsDeadline_IsRefusedWithCode4`). Under up to 1 % of adverse drift every quote still fills, within 2 % of its prediction; above 1 % (up to three trades of 3 % each) the floor refuses about one in four (31 of 133) 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. (Percentages are the table's own ratios, read from its minima: the lowest fill, 9,653 bps, is 3.47 % under its prediction.)

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. These fork figures were measured at 8949a9d on 2026-09-05 (Version 2.3 measurement).

§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`). Measured in CI at 969bcc6; the live-Base figures of the 2.3 are not carried over.

WORLD	`PREVIEWPLAN` MEAN	MIN	MAX	Σ	`PREVIEWANDENCODE` MEAN	Σ
mocks, discovery: the pair known to the factory only, one pool found	148,772	140,519	151,081	1,871	196,957	2,022
mocks, fresh registry: three seeded pools priced	362,856	350,680	365,600	2,952	412,921	3,353

`batchQuote` of ten entries on mocks: 3,299,195 gas, 329,919 per entry. The fresh-registry mock costs more than discovery because it prices three pools where discovery found one; a quote is an `eth_call` an integrator never pays for on chain.

§12.4 The in-frame budgets

The Router's automatic door runs the same search under a fixed budget, so the plan is a function of the state and never of the gas the caller brought. `SOLVE_GAS = 14,700,000` (`Core:2521`) is 2^{24} (the per-transaction cap of EIP-7825, 16,777,216) less about 2.08 M kept for the pull, the execution and the self-call [derived from the code's comment, `Core:2511-2521`]. The Solver's capacity walk runs on what the search leaves, never more than `WALK_CAP = 8,000,000` (`Core:2531`):

$$W \;=\; \min\!\Big(\tfrac{63}{64}\,(S - s - 20{,}000),\;\; \mathrm{WALK_CAP}\Big), \qquad S = \mathrm{SOLVE_GAS} - 1{,}200,$$

with s the gas the search itself spent and 20,000 kept for the splice (`Solver:309-341`). A walk's cost is the venue's choice (each read is capped at `GAS_CAP = 100,000` , `Core:215`), so without a ceiling a pair's held books could burn all of `SOLVE_GAS` and leave the automatic door too little to settle. The code records the two bounds that put the cap where it is: 4,850,000 is the least a live mainnet V4 row's rebuild must be handed (4,800,000 is not enough), and 11,510,000 the most with which four books that burn their whole `GAS_CAP` still settle at the door (11,560,000 does not): `WALK_CAP` sits between them (`Core:2523-2531` , `test/CapacityWalkBudget.t.sol`). A solve entered on less than S, or whose search left less than 20,000 of it, stops with empty revert data (`RouterE(17)` at the automatic door), never with a different plan; the quote door enters on more and hands the walk the same W.

§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 (`Core:426`) 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 (`Router:149` ; computed as `fee - t1` , so the two always sum to the fee with no dust, `_payFee` , `Router:794-803`). The rate and the split are constants. The two treasury addresses are storage: the control key can re-point them (`setTreasuries`) until it renounces control for ever (`renounceCon-`

`trol`), and each payee can move only its own slot (`moveTreasury`), also after renouncement. 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 is one sentence: **charge once, on the first bridge coin the Router holds.** The Router asks one producer, `BPC.feeCalendar` (`Core:439`), which scans the route for the first hop whose input token is a registered bridge, in hop order — that hop is `feeHop` — and charges 28 bps of that hop's *measured* input. On hop 0 the base is the measured pull capped by what the route committed; on a later hop it is the Router's balance of the bridge coin above its foreign baseline *and* at most what the previous hop's legs measured, the quantity already resistant to fee-on-transfer and to stray balances. The Router charges nothing else on that route. A single-hop route whose destination is itself a bridge coin (TOKEN → WETH) charges on the output instead, after the floor check (`feeOnOut`, `paysOnInput`, `Core:448-453`): 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. 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: charging such a route once, on hop 0, reopens 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`). 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 (it charges once per hop when no hop input is a bridge, once otherwise), so the preview and the delivery agree in both regimes (`test/ExhaustionRegimePreviewParity.t.sol`).

The fee is pushed to each treasury with capped gas (`FEE_GAS = 150,000`, `Router:212`), so a treasury address or a fee token that refuses the transfer cannot stop a token from trading. A share the token refuses does not revert the swap: it is booked to its slot (`owed`), stays in the Router, is excluded from every balance a swap reads in that transaction, and is claimed later by the slot's holder (`claimFee`). The fee is charged either way. What counts as "paid" is measured, never taken from the token's answer: the Router's own loss and the payee's gain are both read, the larger witness is what reached the payee, and only the remainder is booked, so a claim never pays a share twice.

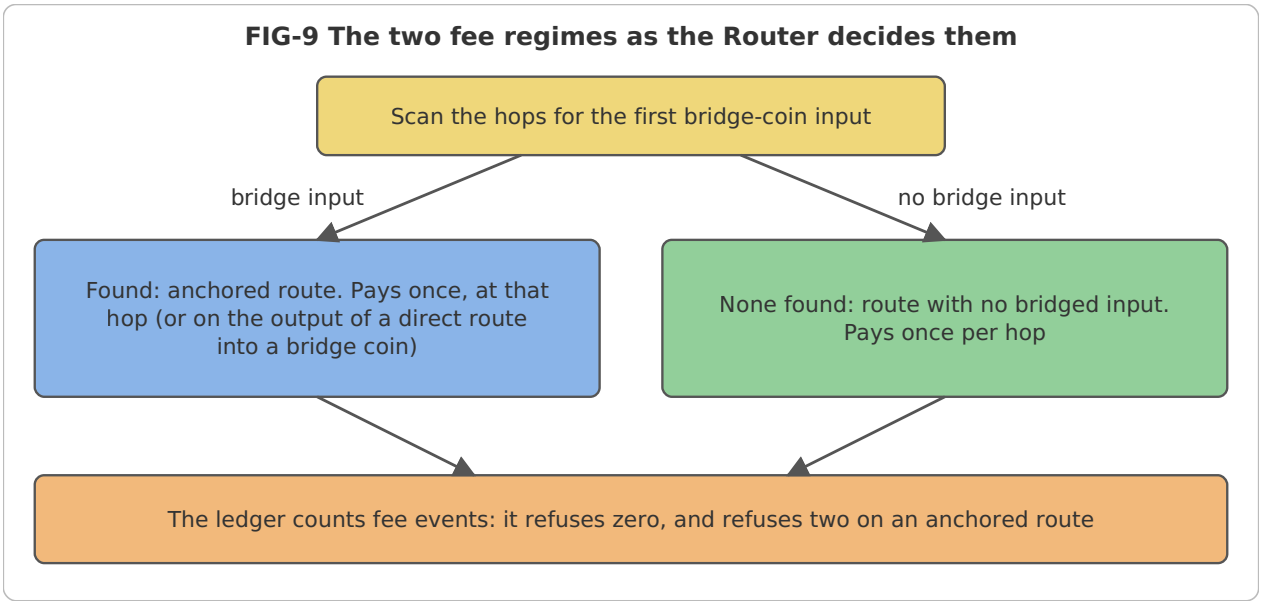

FIG. 9 The two fee regimes as the Router decides them

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` (`Router:687`): one number, one producer. And the fee ledger. `_payFee`, the one function that transfers the fee, counts into a transient slot (`TSLOT_FEE`, `Router:199`); every settlement reads the slot once, refuses a delivery that paid nothing (`RouterE(15)`, `Router:1660`) and, on an anchored route, a second payment (`RouterE(16)`, `Router:1661`), then clears it for the next swap in the same transaction. The claim: a settlement pays the fee, and pays it once where the rule says once. The enforcement site: the Router itself, at run time, on every settlement, on top of the review-time checks of §13.4. A fee at or above the amount it is charged on reverts (`RouterE(8)`), and a fee charged in hop 0 of a cyclic route reduces the route's output cap, saturated, so the cap the floor sees is the cap after the fee.

§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 nothing 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 or two hops; bridge in no, first, middle or last position; one or two legs per hop), fuzzed amounts from 10^{12} to 10^{21} wei, the Router holding nothing after every settlement. The table of shapes is asserted to reach the Router's hop ceiling and no further: a new ceiling turns it red until the shapes of the new depth are written. The fuzzed test runs under the default profile, 256 runs per CI pass (`foundry.toml`).

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

ROUTE SHAPE	REGIME	WHERE THE FEE LANDS	MEASURED
A → W → B	anchored at hop 1	the bridge coin hop 0 produced	one event, ⌈28 bps × what hop 0's pools paid out⌉
A → C → W (bridge only as output of two hops)	exhaustion	each hop's input	two events

How often the fee tests notice a defect is measured rather than assumed (`docs/assurance/fee-seal-detection.json`): each fee mutant — the regime predicate moved either way, the commitment producer, the doubled fee, the fee never charged on the input, the fee charged on both sides — is run under twenty fuzz seeds per fuzzed test and once per deterministic test; the detection rate is reported with a Wilson 95 % interval. The figures below were measured on 2026-09-05 (Version 2.3), before the hop ceiling fell to two.

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

FIG-10 Detection of each fee mutant by each test

	FeeSeals fuzz	Router campaign	Covering array t=2	Junk-prefix escape	Exhaustion preview parity
exhaustion charges hop 0 only	20/20	20/20	no	yes	yes
exhaustion skips hop 0	20/20	20/20	no	yes	yes
commitment counts first leg only	20/20	20/20	no	no	no
fee doubled	20/20	20/20	yes	yes	yes
input-side fee never charged	20/20	20/20	no	yes	yes
fee charged on both sides	20/20	20/20	no	yes	yes

green = detected (20/20 runs, or yes); red = not detected (no)

FIG. 10 Detection of each fee mutant by each test

FIG-10 — Detection rate of each fee mutant by each test, as a heat map: kills over runs with the Wilson interval.

Read across a row: which tests see this defect. Read down a column: what a test can and cannot see. The Router campaign walks two hops and splits hop 0 across two pools, with a bridge coin in the universe, so one campaign holds every regime. The covering array is built to cross the regime boundary, and every one of its rows crosses a bridge; it is the test for the doubled fee, and the other columns are what see the rest. The mutation guard admits a mutant only where a named test kills it, and 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 both are constants with no 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; a settlement cannot pay nothing or pay twice on an anchored route, because the Router counts and refuses; and no token's refusal of a fee share can stop a trade, because the share is booked to its slot and claimed, never lost. The limit: the treasury addresses are storage, re-pointable by the control key until it is renounced and by each payee for its own slot; the rate, the split and the count are not.

Part IV — Security and verification

§14 Security model

§14.1 Attacks and the checks that refuse them

The catalogue below is the founding set of 23 classes, kept in the order the proof was built. The machine-readable catalogue in `docs/assurance/threats.json` has since grown to 93 classes (85 answered by a named guard, 8 out of scope with the reason recorded), and `.github/scripts/assurance/threat_coverage.py` fails the build if a guard symbol no longer exists in `src/` or a named test no longer exists in `test/`. Guards are named by symbol, never by line.

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` (Router:307) → `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)` (Hub:436); 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:478)	Core	blocked	`test/BoundaryEquality.t.sol`, `test/CompilerPanicSurface.t.sol`
Denial of service by unbounded iteration (DASP-5)	every route dimension bounded: `MAX_HOPS = 2` (Core:136), `MAX_LEGS_PER_HOP = 5` (Core:149), so a route holds at most 10 legs, the product of the two and no third constant; `MAX_SLOTS = 16` (Hub:93), `MAX_FACTORIES = 16` (Hub:144); in-transaction search capped at `SOLVE_GAS`	Router `_execute` → `RouterE(3)`; Solver; Hub → `HubE(4)` (factories, Hub:818), `_canInsert` (slots, Hub:2038)	blocked	`test/RouteHopCeiling.t.sol`, `test/Pin01LegCeiling.t.sol`

CLASS (ID)	THE CHECK THAT REFUSES IT	WHERE IT IS ENFORCED	STATUS	THE TEST THAT SHOWS THE CHECK CAN FAIL
Transaction-order dependence / sandwiching (DASP-7)	the Iron-Law floor, hard-clamped at `FLOOR_HARD_MAX_LOSS_BPS = 2,000` (Core:400), enforced independently of the caller's minimum, which may only tighten it	Core `ironFloorBpsShv` (Core:3345); 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_ParityBySplitRoute_AttestedFloorEqualsEnforcedFloor` (`test/FloorParitySolverRouter.t.sol`)
Caller-supplied quantity substituted for a measured one (SOK-FEE-BASE)	the fee base is a measured balance (`_chargeHopFee` , Router:748); `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` , Hub:386) and frozen after renunciation	Hub `addFactory` , `_probe` (Hub:1200)	blocked	`test_C4_FirstPin_AfterRenounce_MustNotAttestTheLiveAnswer` (`test/T19ReadmissionEdge.t.sol`)

CLASS (ID)	THE CHECK THAT REFUSES IT	WHERE IT IS ENFORCED	STATUS	THE TEST THAT SHOWS THE CHECK CAN FAIL
Admitted venue whose logic moves behind a proxy (SOK-PROXY-VENUE)	re-admission after renunciation refused when the runtime moved (`factoryCodehash` , Hub:357); 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`)
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, not the declaration	Router `_recordHits` (Router:2267)	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`)	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` , Core:639); a pool id derives from its own key; `hasCode` and the liveness proof discard what does not resolve	Core; Hub `_probe` , `_scanV4` (Hub:1284)	blocked	`test/ConditionAdequacyCore.t.sol` , `test/KindIsDerivedNotDeclared.t.sol`

CLASS (ID)	THE CHECK THAT REFUSES IT	WHERE IT IS ENFORCED	STATUS	THE TEST THAT SHOWS THE CHECK CAN FAIL
Malicious hook on a hooked venue (SOK-HOOK)	three layers, §14.2: immutable address bits (`hookAltersDeltas` , Core:3369 at `969bcc6` , Core:3425 at `6387350`), allow-list with code-hash pin for any hook that runs in the swap (`hookPaused` , `isHookLive`), the projector in the Solver's candidate merge (Solver: 1556-1590)	Router `_execV4Amt` → `RouterE(9)` (Router: 2033,2049); Hub `allowHook` → `HubE(1)` , `_admitOnRegistration` → `HubE(8)`	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` , Router:1722), and an unknown kind reverts	Router `_execScaled` → `RouterE(8)` (Router:1777)	blocked	`test/regime/HostileVenueMatrix.t.sol`
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 `swapExactInWithPermit2` (Router: 461)	blocked	`test/RouterPermit2OneStep.t.sol`
Governance capture (SOK-GOV)	there is no governance: no token vote, no timelock, no proxy; the control powers (pausing, the role and configuration setters, the fee treasuries, the Router's 48-hour rescue of tokens sent to it by mistake) end for ever with `renounceControl` , and venue admission, which outlives renunciation by design, only grows the registry	—	out of scope	—
Malicious upgrade (SOK-UPGRADE)	the contracts are immutable; there is no proxy and no implementation slot	—	out of scope	—

CLASS (ID)	THE CHECK THAT REFUSES IT	WHERE IT IS ENFORCED	STATUS	THE TEST THAT SHOWS THE CHECK CAN FAIL
Compromise of an external 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	—
Compromise 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 these twenty-three classes are answered by a named guard and four are out of scope with the reason recorded; the later classes in `threats.json` (the quote surfaces, the Gate, the registry clock, the fee push) follow the same two-way rule. The catalogue is a lagging denominator, because it holds only what has already happened to someone, so it is cross-checked from the other direction by an asset closure (`docs/assurance/assets.json`, checked by `asset_closure.py`): 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 (theft, freeze, dilution, misdirection, griefing), thirty cells, each carrying a status. Nineteen cells are covered by named threat classes; every threat id the closure names must exist in the catalogue. The closure's value is that it can name a cell nobody has ever exploited.

§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; 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`, Core:3425 at `6387350`):

$$(\text{uint160}(\text{hook})\ \&\ \texttt{0x3FFF})\ \&\ ((1 \ll 3) \mid (1 \ll 2)) \neq 0 \implies \text{leg refused, } \texttt{RouterE(9)}\ (\text{Router:2033}) \tag{E18}$$

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 and a code pin, for the hooks that run in the swap. The same address answers a second question without a call: does the manager enter this hook during a swap at all? Four bits say so — before-swap (1 ≪ 7), after-swap (1 ≪ 6) and the two delta flags — and the Router reads them with one mask (`hookRunsInSwap`, `0xCC`, Core:3443). The list and the pin are required of exactly those hooks. A hook with none of the four bits is never entered by its own pool's swap, so there is nothing about its code an operator could judge that could change a swap, and the Router admits its pools by the bits alone; quote equals execution for it exactly as for a hookless pool (the comment above Core:3443 states the rule and is the source, no derivation beyond the dispatch-on-bits fact is claimed) . For a hook that does run in the swap, the Hub holds the allow-list (default closed) and pins the hook's codehash at admission (`allowHook`, Hub:543). A hook is *paused* (`hookPaused`, Hub:595) when the curator revoked it, or when it is listed and its runtime has moved since the pin — it is then refused on every door (`RouterE(9)`, Router:2049) and resumes only if re-admitted, with its pools' registry state preserved. After renunciation the curator keeps the power to admit and loses the power to withdraw: `allowHook(h, false)` is refused (`HubE(1)`, Hub:551), and a listed hook whose code moved cannot be re-armed at its new code (`HubE(1)`, Hub:559), so it stays paused for ever, while listing a new hook or re-listing an unchanged one still works. A revoked hook is not re-admitted by registering a pool that carries it (`HubE(8)`, Hub:582); registering a hooked pool is otherwise the admission of its hook (`_admitOnRegistration`, Hub:580).

A hook nobody ever listed is a different case and is stated as one: the explicit door accepts it as the caller's own signed choice, bounded like every venue by the in-frame promise, the attestation gate and `userMinOut`; the automatic door never proposes it, because the registry reads filter rows by `isHookLive` (Hub:621; the filter in `getActivePools`, Hub:1596). 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.

One reach the bits do not describe is stated rather than hidden: a hook that runs in the swap can call the manager on any pool, and a donate, a swap or a liquidity change there enters that pool's hooks. That is bounded by the manager's flash accounting — a delta left unsettled reverts the whole unlock, which the fork tests exercise (`test/fork/NestedHookSettlementFork.t.sol`: a nested donate reaching an unadmitted hook, a take that never settles, an admitted hook that never settles its donate) — by the same floors, and by the admission of the hook that makes the call, never by the bits of the one it reaches.

Layer 3 — the automatic route is never even shown. The projector Ξ removes any pool whose hook alters deltas from the Solver's candidate set before ranking: the filter sits in the merge of registered and discovered pools inside `_rankedPools` (Solver:1531; the two loops at Solver:1584 and :1590), and the Hub's read channel serves only hooks that are live. A route touching a delta hook does not participate in the maximisation, so the planned door cannot assemble a route the Router would refuse. The Router's own check is not removed to make the two agree: it stays as the fail-closed floor. A route-shape rule completes the layer: canonical order, route-wide — no hookless leg ever executes after a hook has run (§11.4).

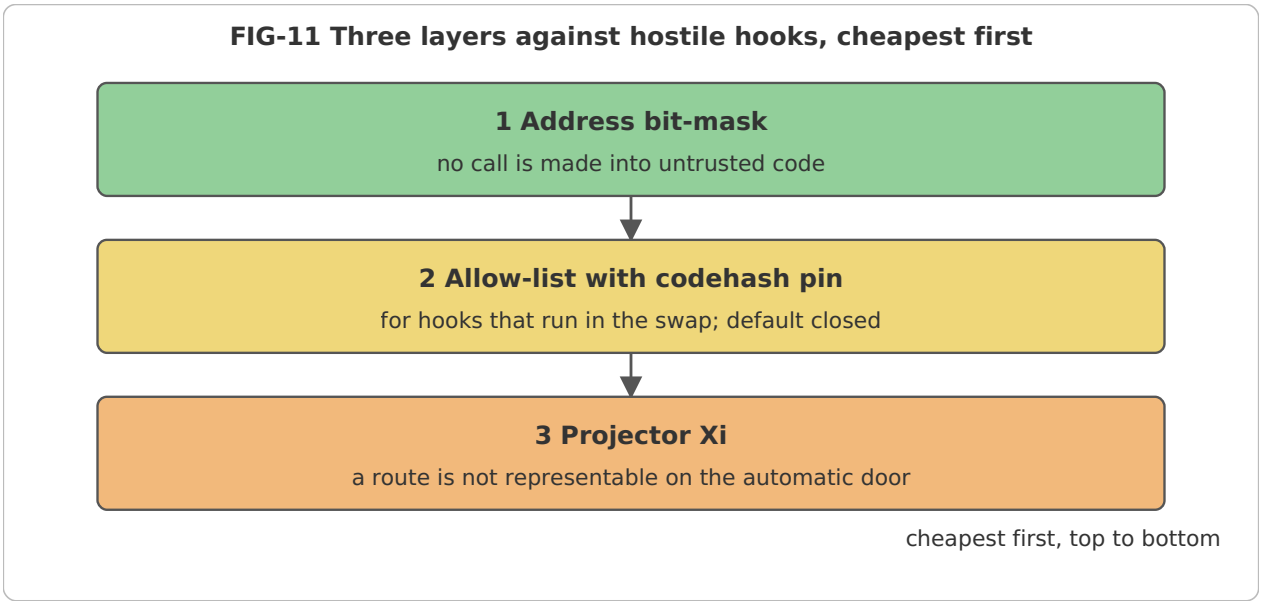

FIG. 11 Three layers against hostile hooks, cheapest first

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 for hooks that run in the swap (default closed), the projector Ξ (route not representable on the automatic door). Each layer answers a different capability; the residual is named.

Defeating the stack requires a hook that carries no delta bits, sits on the allow-list unchanged (or runs in no swap at all), 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. An allow-listed, delta-free hook can still revert mid-swap and waste the gas of trying; that is the named residual of the middle layer. 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` (`onlyControl`, Hub:536), a control power that dies with renunciation, 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 (`ironFloorBpsShv`, Core:3345), 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

Re-run in CI at 969bcc6; every figure matches the 2.3 measurement. With no attacker the victim's gap to the quote is 0.27 %.

The guarantee the test asserts at every grid point is structural, and it is the part that does not depend on the table: 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 (`test_SandwichCurve_TheFloorBoundsTheVictimAndStarvesTheAttacker` , line 132: `assertFalse(settled, "the refusal region is closed upward")` once the first refusal has occurred) — past the edge every larger manipulation is refused and the attacker is left holding the price they moved. 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 the read, burns all gas on the 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. The file holds 20 cells, ten venues times two doors (`test/regime/HostileVenueMatrix.t.sol` , 20 `test_Matrix_*` functions, counted at 969bcc6); the rule for every cell is that the swap settles with the delivered amount equal to the balance delta and nothing left on the Router, or refuses with a selector of ours, with sharper oracles for the pays-nothing and pays-half cells (they must be refused) and for the re-entering pool (the harness records whether its nested swap ever ran). 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. All 20 cells pass in the CI run at 969bcc6. Every guarded read the Core makes to a venue is capped at `GAS_CAP = 100,000` gas (Core:215), so a venue that burns gas on a *read* costs a bounded amount and is dropped, not fatal.

A venue the protocol does not control can also arrive behind a wrapper. The Gate puts a permissioned venue behind an immutable wrapper whose adapter is proposed, pinned by codehash and accepted only after a 48-hour delay (`ADAPTER_DELAY`, Gate:76); a pair holds at most four wrappers (`GATE_MAX_WRAPPERS = 4`, Core:192; the fifth is refused with `GateE(16)`), and a wrapper's quote on the planning path runs under a fixed gas allowance (`GATE_QUOTE_GAS = 500,000`, Core:3271), so one adapter cannot spend the caller's gas. Three classes in `docs/assurance/threats.json` answer the abuse this invites, each with its guard and the test that fails without it: an unbounded wrapper list spending a solve's gas (`GATED-AUTO-LIST-GAS`, guard `GATE_MAX_WRAPPERS`, `test_GatedAuto_WrapperListIsCappedAtFour`); an adapter that over-quotes its venue (`GATED-AUTO-OVERQUOTE`, guard `LEG_FLOOR_BPS`, `test_GatedAuto_OverquotingAdapterHoldsTheFloorUntilCleared`: no funds move, the floor and the caller's minimum hold); and dust swaps that pump a wrapper's use counter (`GATE-DUST-USE`, guard `MATERIAL_SHIFT`, `test_GateVital_DustInputNeverTicks`). The third-party venue's own security is still not claimed (§14.1, SOK-BRIDGE-EXT).

§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(user, tokenOut, quoted, realized, floorUsed, blockNumber)` (Router:272-275, emitted at Router:1697) — the final hop's quote, 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, including you. `user` is the payer and `tokenOut` is indexed, so the record is filterable by either. `floorUsed` is published net of a transfer tax on the output token, in the units the floor is attested in, so on a settled swap with no output-side fee it never exceeds `realized` (the event's own documentation, Router:268-270). 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: hand the admin seat to another address (`setAdmin`); redirect the two fee slots (`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`, `cancelRescue`, `executeRescue`, `RESCUE_DELAY = 48 hours`, announced on-chain by `RescueQueued`, `RescueCancelled` and `Rescued`). Every one of these is gated by one modifier, `onlyControl`, which refuses any caller that is not the admin and refuses everyone once control is renounced (`RouterE(1)`, Router:305).

What it cannot do: move the fee rate or the split, because both are compile-time constants with no setter (`PROTOCOL_FEE_BPS = 28`, Core:426; `TREASURY1_SHARE = 3_000`, Router:149); take principal, because the Router grants no allowance (its own header states "No `approve` anywhere", Router:28), there is no `selfdestruct` anywhere in `src`, and there is no upgrade function; or reach the fees the treasuries are owed

through the rescue path. A rescue returns the Router's balance of a token *less* what is booked as owed to the two fee slots (`executeRescue`, Router:416-420), so even a hostile admin who queues a rescue of a fee token takes only the stray remainder.

One refinement of the 2.3 wording is needed here: the Router does not hold *literally* nothing at rest. Swaps settle or revert inside their own frame and fees are pushed to the treasuries as they are charged, but a push that the fee token refuses (an issuer's blacklist, for example) is booked to `owed` instead of reverting the swap, and waits there for its slot's holder to claim it (`claimFee`, Router:347; register row in `SHARED_QUANTITIES.md`). At the end of a swap the Router holds zero in any token that moves the amount it is asked to move, and in native; the one stated bound is a token that keeps balances as shares and rounds each transfer down, which can leave at most one share per outbound transfer (Router header, R1; `test/RouterBalanceAfterSettlement.t.sol`). What the Router holds at rest is therefore the booked `owed` balance and that bounded crumb, and neither is a swapper's principal.

What a hostile Hub key can do: list a venue or a hook (`addFactory`, `allowHook`) and add a bridge token (`addBridge`, at most three); seed pools (`seedPool`, `addV4`, through an operator seat or the admin itself); pause the registry's learning (`setPaused`); repoint the Hub's view of the Router, Solver and Quoter (`setRoles`); grant or revoke operator seats (`setOperator`); remove a bridge (`removeBridge`); and repoint the V4 manager (`setV4Manager`, the sieve's single lever; the Hub's own comment records what the sieve rests on and that a dishonest manager is bounded, not absent). **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)`, Router:377; `HubE(2)`, Hub:486): 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 (Router:378, Hub:487) 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` and the whole rescue path (`queueRescue`, `cancelRescue`, `executeRescue`). The fee rate and the split were constants and stay so; the wiring (Permit2, WETH) and the pause flag freeze at their current values for ever, and the Router keeps executing swaps under that fixed configuration.

One door on the Router is deliberately left open, and it opens onto the payee, not the protocol. `moveTreasury(to)` has no `onlyControl` modifier: whoever currently holds treasury slot 1 or slot 2 may reassign *that* slot to any non-zero address, also after renunciation (Router:337-342). It exists because a fee token that blacklists a treasury address would otherwise revert every swap of that token in the fee push, and with control renounced nobody else could move the address (the function's own comment, citing bounty #55). What it can do is narrow and stated: the holder moves only the slot it holds (a caller holding neither slot is refused with `RouterE(1)`); the new address cannot be zero (`RouterE(3)`); if one address holds both slots, a call moves slot 1 first; the call emits `Cfg(1, to)` or `Cfg(2, to)`. What it cannot do: change the admin, touch the other

slot, touch the fee rate or the split, or pause anything. Because the *slot*, not the address, is the creditor of any fee booked to `owed`, a slot that has moved claims everything booked to it (`claimFee`, Router:347), so a payee that rotates an address does not strand what it was owed. Earlier versions of this text said the treasuries freeze at renunciation; that was true only of `setTreasuries`, and the contract's own `renounceControl` documentation says so (Router:361-365).

On the Hub, the **control tier** dies: `setRoles`, `setOperator`, `setPaused`, `setV4Manager`, `removeBridge`, hook *de-listing* (`allowHook(h, false)` reverts with `HubE(1)` once control is renounced, Hub:551), and `initialize`, which is also refused after renunciation (Hub:457). The **curator tier** survives, and only grows the registry: `addFactory` (refusing, after renunciation, a re-listing whose runtime codehash moved or whose kind, mode, init hash or fee and spacing set differs from the listed row, Hub:881-887), `addBridge` (add-only, at most three, idempotent: Hub:111, :649-658), and `allowHook(h, true)` (refusing a listed hook whose code moved, Hub:559; re-listing an unchanged hook re-pins it); registering a pool with a revoked hook through `seedPool` or `addV4` does not re-admit that hook (`HubE(8)`, Hub:582). The admin and existing operator seats keep `seedPool` and `addV4`, and because `setOperator` is dead 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, as the Hub's own documentation puts it. 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 a payee moving its own fee slot — 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.

§14.9 The guarantees, each with its own check

Everything in this section reduces to nine guarantees that the repository's README states in one table. Each row names the code that enforces it and the command that turns red if it stops holding; none of them asks to be taken on trust. The table below adds the on-chain refusal and the section of this paper that derives it.

GUARANTEE	ENFORCED BY	CHECK IT YOURSELF	HERE
No swap without the caller's own floor	`RouterE(10)` on every value door when `userMinOut == 0` (Router: 456, :467, :523, :560)	`forge test --match-path test/A1EveryValueDoorRefusesZeroMinOut.t.sol`	§11.7
Delivered output at least `max(protocol floor, userMinOut)`, or the swap reverts	the Router's settlement check, priced by the Core evaluator (`RouterE(5)`)	`halmos --contract CoreFormalGateSpec`	§11.5

GUARANTEE	ENFORCED BY	CHECK IT YOURSELF	HERE
The quote and the enforced floor agree	one evaluator for preview and execution	`forge test --match-path test/PreviewExecutionParity.t.sol`	§12, §15.1
No reentry during a swap, pool callbacks and the output transfer included	a transient (`tstore`) lock spanning the swap (`RouterE(7)`)	`forge test --match-path test/ReentryFromTheOutputTransfer.t.sol`	§11.4
The Router ends a swap holding nothing of what it moved, with the single rounding bound for share tokens stated and pinned	balance-delta settlement	`forge test --match-path test/RouterBalanceAfterSettlement.t.sol`	§11.6, §14.6
The Router leaves no token allowance standing	a static guard over the source	the `Router grants no allowance` step of the CI workflow (`.github/workflows/ci.yml`)	§14.6
The V4 fee is measured and fails closed	`effV4Fee`	`halmos --contract EffV4FeeFormalSpec` ; Certora rules `dynamicUsesMeasuredLpFee` and `dynamicWithProtoFeeFailsClosed` (two of the five rules), specified in `certora/specs/EffV4Fee.spec` (`certora/conf/effv4fee.conf` ; Appendix B.5)	§15.1
Every contract fits EIP-170 with a published margin	size guard in CI (fails at 24,000 bytes of runtime, 576 bytes under the 24,576 limit) and in the suite	`FOUNDRY_PROFILE=release forge build --sizes` and `test/DeployedSizeGate.t.sol`	§16.3
What is deployed cannot change unnoticed	codehash and property pins of the live generation	`forge test --match-path test/fork/DeployedParity.t.sol` (archive RPC)	§14.2, §16.3

Read the table in two directions. From a claim to its evidence: each check is a single command, and a guarantee with no failing command is not on the list. From the evidence to its limit: a check proves the property it names under the conditions it names, and §15.11 states what the whole apparatus does not establish. The ninth row is the one that cannot be run from a clean checkout alone, because it compares the repository with chain state through an archive node.

§15 The verification apparatus, measured

§15.1 The suite in numbers

Every figure carries its denominator and the command that reproduces it. Measured at commit `969bcc6` (CI run 43886865849, 2026-11-16).

WHAT	COUNT	HOW IT IS COUNTED
declared tests	3,517 functions in 515 files; 3,562 executed in CI at 969bcc6 (3,416 suite + 146 fork; 8 skipped, 0 failed)	`grep` over `test/` ; CI
green on the default profile, fork suites excluded	3,408 passed · 0 failed · 8 skipped, of 3,416 tests in 521 suites	`forge test --no-match-path "test/fork/**"`
of which, by kind	54 `invariant_*` · 240 fuzz · 3,122 unit (derived: 3,416 − 54 − 240)	count of `[PASS]` / `[SKIP]` lines by name and by `runs:` in the same run
green on live liquidity	146 passed · 0 failed · 0 skipped, in 48 files of `test/fork/`	`forge test --match-path "test/fork/**"` with an archive RPC
suite plus fork lane	3,562 tests: 3,554 passed · 0 failed · 8 skipped (derived: 3,416 + 146)	the two lines above
stateful invariants	54 `invariant_*` functions in 18 files	`git grep 'function invariant_' -- test`
symbolic and formal	15 Halmos checks reporting `[PASS]` at `6387350` (Appendix B.3) · 4 Certora specs (specified, Appendix B.5)	`halmos --contract <name>` ; `certora/conf/`
curated mutants	1,895 registered in `M` , plus 19 in `M_FORK` ; 1,753 run on this commit, all killed	`.github/scripts/mutants.py` ; `check_targets.py`
regime covering arrays	strength 2: 60 rows · strength 3: 145 rows; 237 pairs and 1,441 triples in the arrays	`test/regime/RegimeCoverage.t.sol` , `RegimeCoverageT3.t.sol`
shipped sizes (runtime bytes; EIP-170 limit 24,576; project gate 24,000)	Hub 23,360 · Router 23,710 · Solver 21,253 · Quoter 19,760 · Core 21,801 · Gate 10,710	`FOUNDRY_PROFILE=release forge build --sizes src`

Reading the table.

- **Fuzz and invariant settings are fixed, not tuned per run.** The fuzz seed is `0xb1a2ef00` , fuzz runs are 256, invariant campaigns run 50 sequences of depth 50 (`foundry.toml`); nine fuzz tests override the run count in their own file (seven at 64, two at 32). A failure therefore reproduces from the seed.
- **Profiles.** The suite runs on the default profile. The default and release profiles share every compiler option (solc 0.8.36, cancun, `optimizer_runs` 200, `via_ir` , `bytecode_hash = none`), and `profile_parity.py` pins that equality; the release profile itself is what the size guard builds. This paper does not claim a suite run on the release profile.
- **Skipped is not passed.** The 8 skipped tests are counted out of the 3,408: five are fuzz tests of the n-version lane, declared with 0 runs, and three are unit tests.
- **Covering arrays.** The regime matrix is exercised by a strength-2 array (every pair of factor levels appears in some row) of 60 rows and a strength-3 array (every triple) of 145 rows. The assurance documents still quote an earlier generation of the arrays, built when the Router still accepted a third hop (before 2026-11-06); with `MAX_HOPS = 2` (Core L136 at `6387350`) the factor is gone and the arrays above are what runs.

- **Sizes.** At `969bcc6` the margin to the EIP-170 limit is 1,216 bytes for the Hub and 866 for the Router; the CI guard fails a build at 24,000 bytes of runtime, 576 below the limit. The full 16-contract table, measured at the later commit `6387350` (Hub 21,780, Router 22,717), is in §16.3.

§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. The first rule of the repository's contributing contract is red before green: a change that fixes a defect arrives with a test that fails without it, and the test header names the commit at which it was red. A guard without a mutant does not merge.

The mutation guard is a fixed, versioned register, not a generic mutation tool: 1,895 entries in `M` and 19 in `M_FORK`, each pairing one exact line of source, a replacement, and the single test that must fail once that line is altered. It deletes a guard, flips a comparison at the bound that decides a refusal, widens an authorisation, or swaps an error code for its neighbour's. Three properties make it a guard rather than a score.

- **The paired test is green first.** `baseline()` runs the suite on the unmutated tree, and a mutant whose paired test is already red, or never ran, is reported as a failure of the guard, because a test that is red for its own reasons would "kill" every mutant pinned to it for free.
- **A mutation that changes nothing is inert, not killed.** `artefact_fingerprint()` hashes the deployed bytecode before and after; a source edit the optimiser folds away yields a byte-identical artefact and is reported as inert.
- **A target that moves is caught.** `check_targets.py` is a static check, run on every push, that every mutant still points at exactly one line of source.

Killed: all 1,753 run on this commit. The figure is adequacy against this register: a kill count is a floor on what the suite notices, never a ceiling on what could be wrong.

§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 invariants 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` holds the matrix, with the exact edit). It was taken on the 40 `invariant_*` functions in 14 campaigns of that date; the suite now holds 54 in 18 files (§15.1), and the map below is that first measurement, not a re-count on the current set.

The map, mutant to the invariants that die to it.

MUTANT	WHAT IT REMOVES	NOTICED BY
the final transfer pays one wei less	holds-nothing	`RouterHoldsNothing`, `routerHoldsNothing`, `HoldsNothingBeyondTheSeed`
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`, `LedgerConservationA`
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`
`MAX_SLOTS` becomes 17	the registry bound	`neverExceedsMaxSlots`
the input-residual sweep ignores its baseline	stranded money	`StrandedMoneyIsNeverSwept`, `HoldsNothingBeyondTheSeed`

Each mutant above became an entry in the mutation guard, paired with the invariant that dies to it.

§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 or 2, the routes the Router accepts under `MAX_HOPS = 2`), 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. That is 3,456 combinations (`docs/assurance/regimes-covering.json`).

`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 is 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 IN THE ARRAY	SETTLED	REFUSED, OURS	NOT CONSTRUCTIBLE	THIRD WAY
2	60	237 pairs	53	1	6	0

STRENGTH	ROWS	TUPLES IN THE ARRAY	SETTLED	REFUSED, OURS	NOT CONSTRUCTIBLE	THIRD WAY
3	145	1,441 triples	138	7	0	0

The outcome columns are measured by running `test/regime/RegimeCoverage.t.sol` and `RegimeCoverageT3.t.sol`, and a row is counted as passing only if its single assertion holds; the 60 and 145 rows pass in the suite run of §15.1 (60 of 60, 145 of 145). The rows the fixture cannot build are printed by name and count against the denominator. At 969bcc6 the six it cannot build (strength 2) are all pull-only fee-on-transfer rows at six decimals; at strength 3 every row is built. The split is in the table.

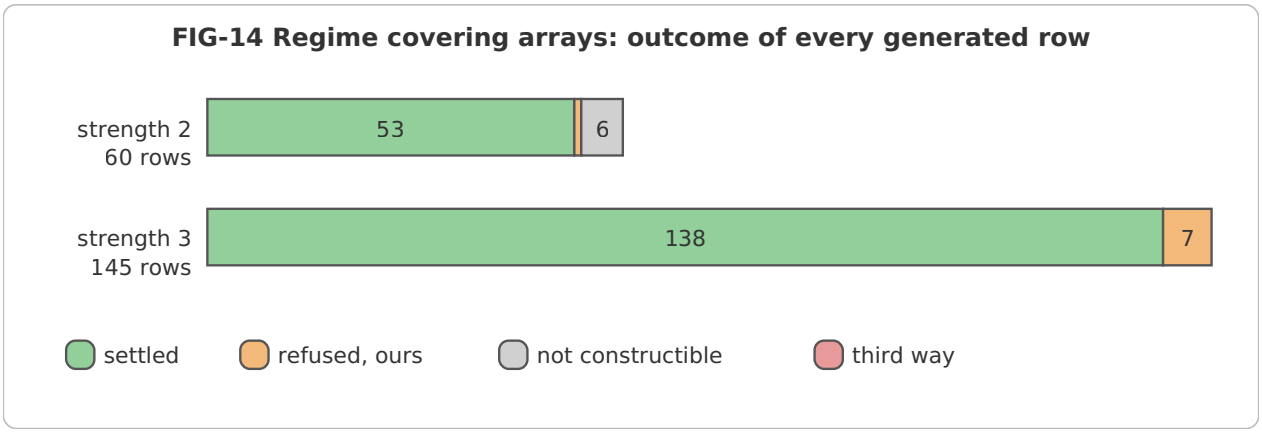

FIG. 14 Regime covering arrays: outcome of every generated row

FIG-14 — Regime covering arrays: outcomes of every generated row at strength 2 and strength 3 — settled, refused with a selector of ours, not constructible, third way — over the combinations of ten factors.

The families not in the array are stated in `docs/assurance/regimes-covering.json`: V4, native V4, Algebra, the native door and hooks, whose harness mocks live in individual test files. 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 $[\text{attested}, \text{attested}/(1-\text{fee})]$, one factor per charging hop, when it comes off the input: the lower edge because a curve concave through the origin delivers at least the net share of the gross output, the upper edge the floor read back on the gross output, which the enforced floor never exceeds. Both frames are asserted on every row. `covering_array.py --check` fails if a committed array has drifted from the factors it claims to cover.

§15.5 The hostile-venue matrix

Token pathologies (fee-on-transfer, rebasing, blocklists, pausing, odd decimals) have long had their tests. Venue pathologies did not have a matrix. `test/regime/HostileVenues.sol` holds one misbehaviour per venue, and `HostileVenueMatrix.t.sol` crosses each with the calldata door (`EXACT_IN`) and the solving door (`BEST`) under the covering array's rule: the swap settles with the delivered amount equal to the recipient's balance delta and nothing left on the Router, or it is refused with a selector of ours.

VENUE	WHAT IT DOES	ORACLE BEYOND THE RULE
lying V2 pair	takes the input and pays nothing	must be refused

VENUE	WHAT IT DOES	ORACLE BEYOND THE RULE
half-paying V2 pair	pays half	must be refused
returndata bomb	answers the reserve read with a returndata bomb	the rule
gas bomb on the read	the reserve read burns all gas	the rule
gas bomb on the swap	the swap burns all forwarded gas	the transaction reverts whole and the user's balance is untouched: no caller can decide this case for the callee
gas bomb on `decimals()`	a token whose `decimals()` never returns	the rule
double callback (V3)	fires the payment callback twice	the rule
re-entrant (V3)	re-enters the Router before paying	the pool records whether the Router let the nested swap run; it must not
`slot0()` reverts (V3)	the state read reverts	the rule
wrong-pair factory	its `getPair()` answers with a pool on other tokens	nothing settles through a pool on other tokens; discovery must not list it

Ten venues times two doors: 20 test functions, `test_Matrix_<Venue>_ExactIn` and `test_Matrix_<Venue>_Best`, all in the suite run of §15.1. Each cell prints `CELL <venue> <door> <outcome>` (settled, refused with our selector, or admission refused), so the outcome of every cell is read from the run and not asserted from memory. At `969bcc6`: six cells settle, ten are refused with a selector of ours (`RouterE(5)`, `(6)`, `(8)`, `SolverE(5)`), two are refused at admission (`HubE(4)`), and two exhaust the forwarded gas and revert whole with the user's balance untouched. A cell passes by settling or by a refusal with a selector of ours, never a third way.

The last row is what the matrix's first run changed. Discovery listed a pool that a curator-admitted factory answered with, and the Solver planned it; the executor refused it at the seam that pays (`LEG-01`), so funds were never at risk, but the pair stayed refused for as long as the impostor won the split. Discovery now proves an asked pool's own pair before listing it, and the cell asserts that `discoverFor` never returns the impostor (§7.2).

§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, not from the Core, and fuzzes the Core against them: constant product with the fee on the input (Uniswap V2), the single-tick square-root-price step with the pool's own against-the-trader rounding (Uniswap V3), and the stable curve $x^3y + xy^3 = k$ solved by Newton's method (Solidly). The direction is asserted first: the Core never promises more than the venue's own mathematics delivers, because a quote above the truth is the defect that reaches a user, while a quote below it is a worse price the floor still bounds. Tightness is asserted second, to the width the rounding order can explain.

FAMILY	DIRECTION (ASSERTED IN THE TEST)	TIGHTNESS (ASSERTED IN THE TEST)
Uniswap V2	`core ⩽ spec`	`spec - core ⩽ 1` wei
Solidly stable	`core ⩽ spec + 2` wei	`spec - core ⩽ spec / 10^6 + 4`
Uniswap V3	`core ⩽ spec + ulp`, with `ulp = L / 2^96 + 1`	`spec - core ⩽ spec / 10^9 + 4`

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 $\sqrt{P}$, which is worth $L/2^{96}$ wei of output; for any pool with $L < 2^{96}$ that is below one wei, plus the one wei the bound carries. The assertion is the bound, not a sample. The stable curve is the loosest, because Newton's method stops within a wei of the root on either side and the Core may carry its own margin. The test also prints the worst gap it observes on a fixed grid for each family (`test_Grid_TightnessIsMeasured`). Each fuzzed oracle runs at the suite's default budget, 256 inputs per test. 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 fifteen test functions over the Core's quote maths (seven relations, MR1 to MR7, instantiated per curve) and `test/RouteMetamorphicRelations.t.sol` holds eight over the Solver's plan, on a two-pool universe with a real Hub and a real Solver and no Router: the relations are about routing decisions, and settlement has the covering arrays of §15.4.

LEVEL	RELATION	WHAT IT SAYS	BOUND THE ASSERTION STATES
Core, V2 · V3 · stable	MR1 monotone	more in, never less out	none (`⩽`)
Core, V2 · V3 · stable	MR2 sub-additive	splitting an order across the same pool never gains	whole `⩽` parts + 2 wei (V2, stable); + 2 wei + 2 ulp (V3)
Core, V2 · stable	MR3 no round trip	in, then back on the updated reserves, never returns more than went in	none (`back ⩽ a`)
Core, V2 · V3 · stable	MR4 scale equivariance	scaling order and pool together scales the output, to rounding	V2: at most k wei over, 1 wei under; V3: $2k + 1$ + ulp; stable: k + a millionth of the output
Core, V2 · V3	MR5 fee monotone	a higher fee never pays more	none (`⩾`)
Core, V3	MR6 direction symmetry	at price 1.0 the two arms of `outV3` agree	two ulps + 2 wei
Core, Solidly	MR7 identity	the volatile arm *is* `outV2`, one producer	equality
plan	MR-R1	the split never pays less than the best single pool would	none (`⩾`), over pools the band keeps
plan	MR-R1d	a dust pool is not quoted, and dropping it costs at most the price gap	`got * 10_401 ⩾ best * 10_000`

LEVEL	RELATION	WHAT IT SAYS	BOUND THE ASSERTION STATES
plan	MR-R2	monotone in `amountIn`	none ($\leqslant$)
plan	MR-R3	adding a pool inside the price band never lowers the plan	none ($\geqslant$), over pools the band keeps
plan	MR-R4	registration order moves the plan by at most one weight unit of the split	`fwd / 10 000 + 16` wei
plan	MR-R5	the attested floor never exceeds the expected output, and is not zero	`floor ⩽ expected`, `floor > 0`

Three of the bounds (MR6, MR-R1d and MR-R4) were measured before they were written, and the number in the assertion is the mechanism, not the sample; the domain limits of MR-R1 and MR-R3 were set by measurement in the same way.

- **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: the price moved one step, worth that much output), while the other quoted zero, its price step rounded to nothing and failing closed. The relation states two ulps, one per arm. Both arms have since been made to round against the trader (commit `e5ced25`); the bound was kept.
- **MR-R1 and MR-R3.** Before the price band existed, the file's header records that with one pool priced 10^{10} away from the other the plan paid 35 % less than the outlier alone (MR-R1); the relations now run inside the band's domain, so that figure is history, not a property asserted at 969bcc6. The relations therefore hold over pools the band keeps. Since the band became the band of no arbitrage (2026-10-28, §10.3), the two prices stay within ±0.5 % of each other in these two relations, and the two depths within $1 : 10^4$. The header records the measurements that set those limits: over ±4 %, MR-R1 met a plan 1.9 % under the pool the band had dropped, and MR-R3 a pool whose arrival lowered the plan 2.7 % (the deeper pool set the median and the cheaper one left).
- **MR-R1d.** A pool holding under 25 ppm of the rest of its pair (`MIN_SPLIT_IMPROVEMENT_PPM` , §10.4) is dropped before it is quoted. Measured on the CI seed before the domain was written: a pool at 7.7 ppm of the other, priced 3.97 % better, would have paid 309 bps more on an order of 0.85 % of its reserve, and a deep pool added next to a 5 ppm one replaced it at 311 bps less; both plans were the deep pool alone, to the wei. MR-R1d pins that cost: never more than the price gap.
- **MR-R4.** A pool whose depth weight floors to the minimum (1 of 10,000 against the deepest) is kept as a dust leg when registered first and cut when registered second. Both plans are sound; they differ by what one weight unit earns in one pool versus the other, observed at 3×10^{-7} of the output inside the ±4 % domain. The bound is that weight unit plus the split's floors, and it is written where it was found.

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

The suite executes one binary, the one the default profile emits, and `profile_parity.py` keeps that profile identical to the release profile (F:146). 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, the Math library's band selection and the Router under three more profiles: `nver1` (`optimizer_runs` 1), `nver2` (`optimizer_runs` 20,000, under solc's built-in Yul step sequence, without which it runs out of stack there) and `nver3` (solc's built-in Yul step sequence, with the function specialiser that the release profile drops),

everything else equal to the release settings. `NVersionCore.t.sol` deploys the shipped binary beside the alternates and asserts equality on fuzzed inputs: `outV2`, `outV3`, `outSolidly`, `outSolidlyStable`, `mulDiv` and `mulDivUp` value by value, and the band selection byte for byte, the whole return or the whole revert. In CI each fuzzed comparison runs 2,000 inputs (`ci.yml`, the N-version lane).

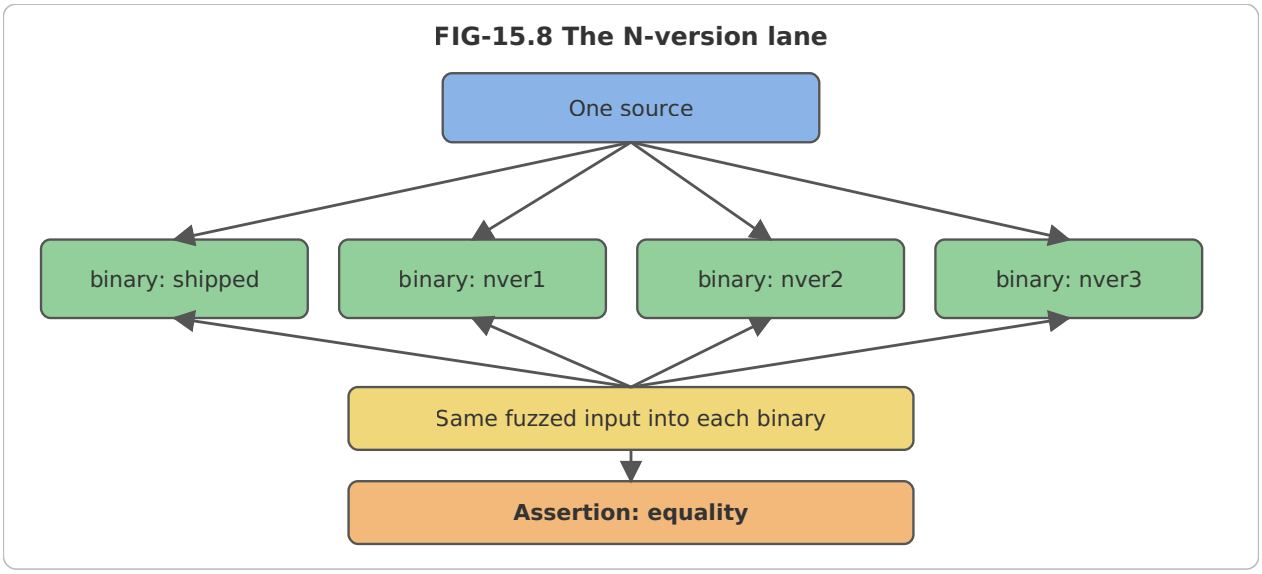

FIG. 15.8 The N-version lane

FIG-15.8 — The N-version lane: one source, four binaries (shipped, nver1, nver2, nver3), the same fuzzed input into each, and one assertion, equality.

The lane cannot pass by comparing a program with itself. Its first test asserts that each alternate differs from the shipped build (modulo the 20 bytes of the linked library address, so a profile that changed nothing is not called "different"), and a missing artefact skips the test unless `NVERSION_LANE=1` turns the skip into a failure. In the default run of §15.1 (at 969bcc6) the lane's five fuzz tests were skipped (F:142: five skipped fuzz tests with zero runs); it runs in its own job, after the alternates are built.

§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 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. Run on 969bcc6, the screen reads 134 refusal sites: 112 at distance 0, 18 at distance 1 and 4 unbounded (an adapter's, a hook's, a factory's and the maths library's code hash). The 2.3 counted 99, 86, 11 and 2 on the tree of 2026-09-05.

A run of the script on `origin/main` at `969bcc6` (this run) lists four sites as unbounded, all of them `code-hash` comparisons: the Hub's hook pin (`Hub:559`), the Hub's factory pin after renunciation (`Hub:887`), the Gate's adapter pin (`Gate:177`) and the Solver's check of its linked Math library against its own runtime code (`Solver:202`). The last two are screen hits rather than open residuals: the Solver compares against code

compiled into itself, and the Gate pins an adapter behind an immutable wrapper (§14.4). The two that matter are the hook pin and the factory pin, 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 register of shared quantities (`SHARED_QUANTITIES.md`, §2.5) enumerates every quantity with more than one producer or consumer, states the *question* it answers, and grades what binds the copies, from `SINGLE` (one producer, nothing to disagree) to `PINNED` (a named test ties the copies), with a CI rule that demotes a row whose test does not reach what it claims to pin. At `969bcc6` the DEX register has 131 rows in five sections and the staking register (`9958d5f`) 37 rows in three (F:101, F:112). Every two-producer quantity in the relational register is tied by a named test: 11 of 11 (`docs/assurance/relational.json`).

RE-GISTER	ROWS	SECTIONS	SOURCE
DEX	131	Floors and fees; Registry state; Identity and keys; Transient state; Measurement harness	`SHARED_QUANTITIES.md` @969b-cc6 (F:101)
Staking	37	Rates and utilisation; Curves; Accounting and identity; Reward and interest projection; Time boundaries	`SHARED_QUANTITIES.md` @9958d5f (F:112)

And the compiled artefact is read, not only the source (`bytecode_invariants.py`): opcodes that must never appear in the contracts (`SELFDESTRUCT`, `CALLCODE`, `ORIGIN`); `DELEGATECALL`, `SSTORE` and `TSTORE` counted rather than forbidden, so that a second delegation, an extra write or a persistent slot where a transient one was meant is visible as an event; and the transient opcodes still present in the executor, so the reentrancy lock has not quietly become persistent storage. For one recorded honest V2 swap replayed on the release artefacts at 969bcc6, 4,948 Router instructions, 426 Hub instructions and 1,078 Core instructions ran with zero opcode mismatches: 32.1 %, 2.7 % and 7.3 % of the three code sections. The bytecode invariants hold on all 15 artefacts the check reads, every declared function among them: the sixteen of §16.3 less `BlazePhoenixGate-Wrapper`, which shares its source file with the Gate and which the check's filter skips. The 88.3 % lower bound on executed instructions was measured at 84d1553 (2026-09-04, Version 2.3 measurement). What they mean: the complement of an executed fraction is 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 a fixed fuzz budget per seed.** A test that misses at that budget may catch at a larger one. Here: the default budget, 256 runs per seed, over 20 seeds (measured 2026-09-05).
- **Some lanes are not in the default run.** The N-version lane skips unless its alternates are built (§15.8), and the fork suite needs an archive RPC (146 tests on five chains: Base, Arbitrum, Optimism, Robinhood Chain and Avalanche; F:142); a green default run says nothing about either.
- **This repository is V2, and V2 is not deployed at the time of writing** (§19).
- **The repository's own list of limits is the boundary of every claim above.** `docs/KNOWN_ISSUES.md` records what is by design (for instance, a route the caller writes is the caller's intent: the Router measures every leg and enforces the floors and the caller's `userMinOut`, and does not refuse a consistent venue for being a worse price), what is an accepted and documented limit (for instance, `canExecute` does not detect tokens that refuse contract recipients), and what is out of scope (read-only figures that cannot change what a user signs). A finding is judged against that file first; what it says is not repeated here, and a statement in this paper never widens it.

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 inside comments is the standing example, and `projection_distance.py` now strips comments before it classifies. 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.

The artefacts are in the repository: a curated mutation guard with per-mutant killing tests, a shared-quantity register, invariant-aimed mutants, a strength-3 covering array, N-version quote maths, a quote-ageing curve and fee-mutant detection rates with confidence intervals. If you know a prior public instance of any of them, we would like to know.

Part V — Architecture, performance and deployment

§16 Architecture and sizes

Plain: five core contracts and three supporting pieces, one of which is pure mathematics imported by the others, and no piece of mathematics is defined twice. *Precise:* the Core is a stateless library with no storage write; Hub, Solver, Router and Quoter import it and none redefines a primitive, so one review of the mathematics covers every consumer; the Gate puts every external venue behind an immutable wrapper, and Math is a stateless library whose codehash the Solver pins.

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

The DEX source tree has sixteen deployable artefacts. Five are the core: Router, Solver, Hub, Quoter and Core. Three are supporting: the Gate and its per-venue wrapper, and the Math library. Eight are adapters that translate one external venue family into the Gate's single interface: AlgebraIntegral, BalancerV2, BalancerV3, CurveCryptoNG, CurveNG, InfinityBin, InfinityCL and LB (`src/adapters/` at `6387350`). Only the venues the repository's README lists as supported are claimed in this paper (Appendix E); an adapter that exists in the source but is not listed is not a claim.

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` in its source	19,303
Math (library)	nothing: the stateless per-hop choice	the Hub, read-only	never	9,768
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	21,780
Solver	nothing, `view` only	the Hub (registry reads, discovery), pools through the Core, the Math library (its codehash is checked against the Solver's own record of it)	never	21,349
Router	admin, treasuries, `permit2` , `weth` , `paused` , `controlRenounced` , rescue queue; **no token balance at rest** beyond the recorded `owed` (the part of a fee a recipient refused)	the input token, the pools of the route, the output token, the Hub (`isBridgeToken` , `isHookLive` , `recordSwap`), the Solver (`findBestRoutePlan` only, `view` , from the `swapBestExactIn` door), WETH and the PoolManager for V4, Permit2 for its door, the treasuries, the recipient, and external venues only through a Gate wrapper; never a caller-supplied target	its own configuration; the one external write per swap is `Hub.recordSwap` on the success path	22,717
Quoter	nothing	the Solver (`view`), the Hub (`isBridgeToken` , read-only), pools for the Exact Pass dry-run (called and reverted)	never	19,892
Gate and **GateWrapper**	the curated list of adapters (codehash pinned) and, per external venue, one immutable wrapper	the adapter, read-only for a quote and as the settlement path for a swap	on admission or replacement of an adapter, after `ADAPTER_DELAY` (48 hours); at most four wrappers per pair	10,758 and 4,414

CONTRACT	HOLDS	MAY CALL	WRITES PERSISTENT STATE	SIZE (RUNTIME BYTES)
Adapters (eight)	nothing	one venue family each	never	3,014 to 6,347 each; 37,340 together

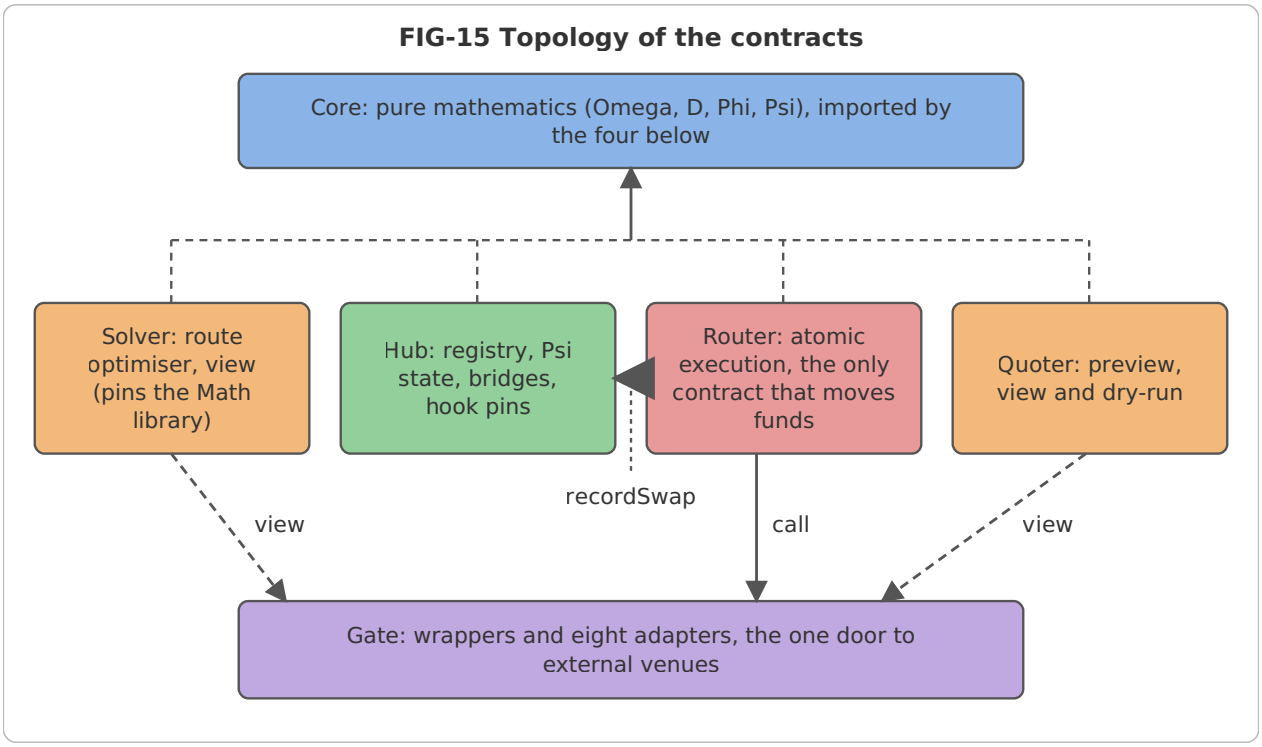

FIG. 15 Topology of the contracts

FIG-15 — Topology of the contracts: 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, with the Math library it pins), Router (atomic execution, the only contract that moves funds) and Quoter (preview, view and dry-run); the Gate with its wrappers and eight adapters as the one door to external venues. 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. An external venue is reached only through a Gate wrapper, so what the Gate trusts is a codehash and a 48-hour delay, not a name (§14.4).

§16.2 The life of one swap

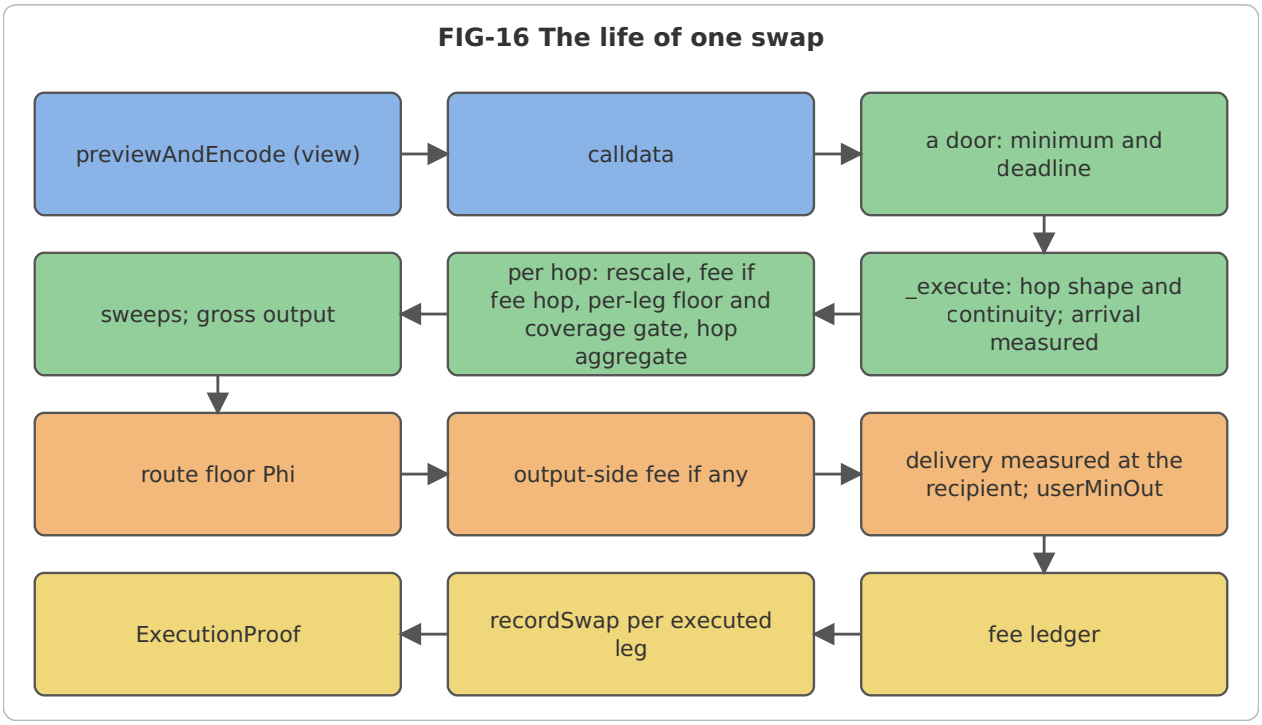

FIG. 16 The life of one swap

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 (an external venue, kind 9, only through its Gate wrapper, §14.4; `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

Runtime bytes of the sixteen artefacts, from the `forge build --sizes src` output of the CI size guard at commit `6387350` (release profile, solc 0.8.36), identical byte for byte to the repository's `.sizes-snapshot` and README table:

CONTRACT	RUNTIME BYTES	UNDER THE 24,000-BYTE GATE	UNDER EIP-170 (24,576)
Router	22,717	1,283	1,859
Hub	21,780	2,220	2,796
Core	19,303	4,697	5,273
Solver	21,349	2,651	3,227
Quoter	19,892	4,108	4,684
Gate	10,758	13,242	13,818
Math	9,768	14,232	14,808
GateWrapper	4,414	19,586	20,162
Five core contracts together	105,041		
Eight adapters together (each between 3,014 and 6,347)	37,340		
All sixteen artefacts	167,321		

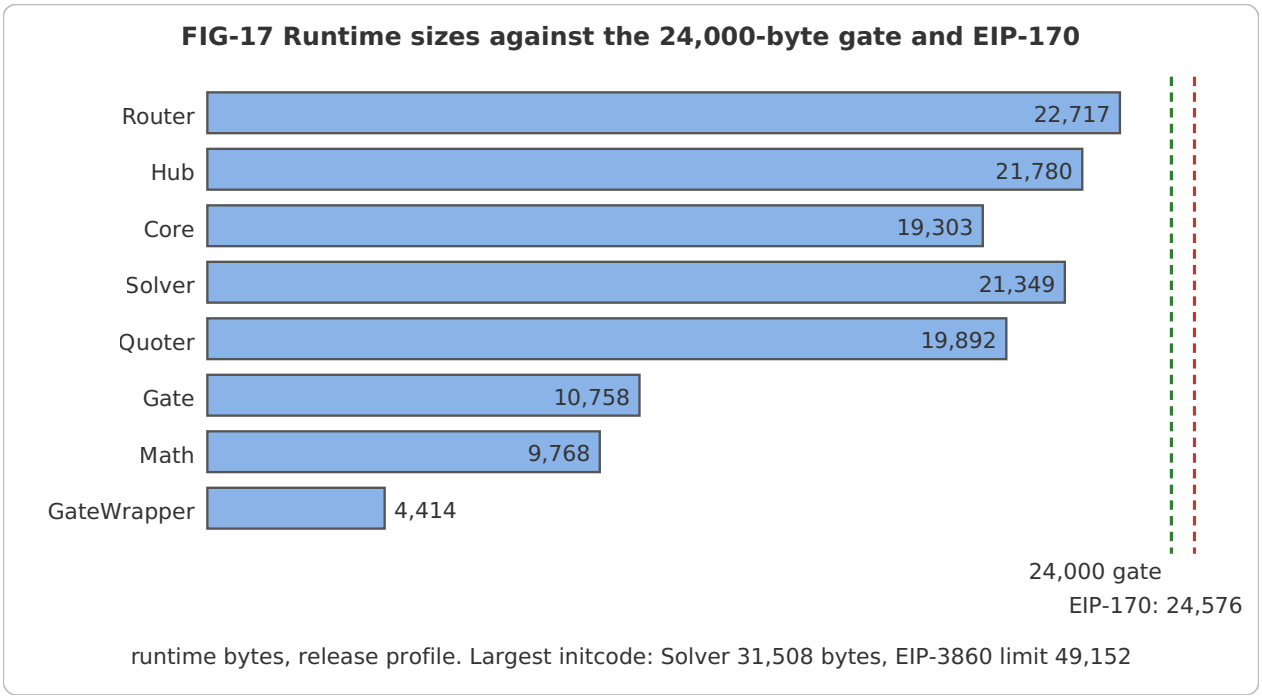

FIG. 17 Runtime sizes against the 24,000-byte gate and EIP-170

FIG-17 — Runtime sizes of the contracts against the project's 24,000-byte gate and the EIP-170 limit of 24,576 bytes, release profile; the largest initcode is the Solver's, 31,508 bytes, against the EIP-3860 limit of 49,152.

Against the 2.3 measurement the Hub is 1,868 bytes smaller and the Router 1,064 smaller, while the Solver is 1,663 larger, the Quoter 8,463 larger and the Core 12,861 larger (derived from the 2.3 table). The CI guard fails a build at 24,000 bytes of runtime, 576 below the EIP-170 limit, and it passed (`All BlazePhoenix contracts under the 24000-byte margin. OK.`). The gate is also asserted inside the suite (`DeployedSizeGate`) on the profile the suite runs under, which `profile_parity.py` keeps identical in compiler options 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. The Router and the Hub sit within about two to three thousand bytes of the wall (1,859 and 2,796 under EIP-170), so every byte added is paid for by a byte removed.

To reproduce: `forge build --sizes src` with `FOUNDRY_PROFILE=release` on the commit named above, and compare against `.sizes-snapshot` . The STK contract is measured separately: runtime 24,157 bytes, 419 under EIP-170, solc 0.8.28, default profile at `c0366cf` ; it is not one of the sixteen.

§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 2 hops × 5 legs = 10 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: 25 rows at `969bcc6` , counted by class below (a row is a field at one site, so a field can appear in two rows).

CLASS	ROWS	MEANING	EXAMPLES
confirmed	17	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, or the callback authenticates on it); `leg.kind` where execution runs the branch it names; `route.singleOutFloor` (monotone: it can only tighten the caller's own floor); the advisory route fields (`totalOut`, `estGas`, ...), read by nobody
declared	3	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 a ceiling and used by quote and execution alike, so the two cannot diverge); `leg.hooks` on non-V4 legs (feeds only the hookless-before-hooked ordering; a lie costs the caller their own route); `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 (`docs/AUDIT_METHOD.md:74-79`). The Hub confirms `kind`; the Router confirms the amounts, the tokens and the concentrated fees; the derivation confirms the V4 fields by construction. The 2.3 text counted 23 fields (14 confirmed, 5 steering, 4 declared); the 969bcc6 file has 25 rows (17, 5, 3). The class of each row was read from `fields.json`; whether the 2.3 count of fields and this count of rows measure the same thing was not checked.

What the Router holds between transactions. No token balance at rest: `INV-1`, asserted by `invariant_RouterHoldsNothing`, `invariant_routerHoldsNothing` and `invariant_HoldsNothingBeyondTheSeed` across the campaigns, each with its guard mutants (Appendix B). Fees stream to the treasuries inside the swap; residuals return to the payer inside the swap. The register refines two edges of "nothing": the Router records `owed`, the part of a fee a recipient refused (since 2026-10-21), and for share-based tokens that round down the final balance is below k + d/n shares after k transfers, zero for tokens that move the exact amount (`SHARED_QUANTITIES.md:54-55`, `test/RouterBalanceAfterSettlement.t.sol`). A mis-send is the only other thing that can sit in the Router: the 48-hour rescue exists to return it, and every swap's baselines exclude it from payout. Core, Solver and Quoter write no storage.

Whom a swap calls. The counterparties of one swap form a closed set of classes: the input token; the output token; the pools the route names (at most 10, the product of the two route bounds); the Hub; the Solver, only through `findBestRoutePlan` from the `swapBestExactIn` door 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; and, for an external venue, its Gate wrapper. 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.

Who holds power. The control functions of the Router and the Hub are enumerated in §14.6 and §14.7 with what each 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, existing operator seats, and each treasury's power to move its own slot (`moveTreasury`) and claim what that slot is owed (`claimFee`). The one ad-

ded power is the Gate's: adding or replacing an adapter waits 48 hours, and a pair has at most four wrappers. There is no `selfdestruct` and no proxy in the Router source (a `grep` for `selfdestruct` and `delegatecall` at `6387350` is empty), and no initialiser after the Hub's one-time `initialize`.

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	25 rows in the register, 17 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	none
token balance at rest in the executor	0 (`INV-1`), apart from the recorded `owed`	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	none
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	167,321 in sixteen artefacts, 105,041 of them in the five core contracts, every one under EIP-170	the settlement contract, plus a server whose size in auditable lines is unknowable	none

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, with real venues and real reserves deposited by strangers, and every number carries its date and the commit it was measured at. *Precise:* a number appears here only if a harness in the repository, or a CI log, 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 deploy-

ment record is §19. This section has two layers and they are not mixed: the measurements of the 2.3 paper (August to September 2026), kept with their date; and the newer measurements of the DEX at `969bcc6`, taken from the CI run of 2026-11-16 (job numbers in the text) and from the gas campaign of 2026-11-13. Where the newer run did not repeat an old measurement, the old number stays and says so.

§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 (2.3, 2026-08). 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, surviving each family's callback conventions, Algebra's included. Both identities held on all four chains, and the differences column the table would need is empty. This paragraph carries the Version 2.3 measurement (2026-08).

§17.2 Preview to delivery fidelity

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.

2026-08-24 (2.3). The band was 0 to 1 bps on every executed pair on all four chains, with the Solver matching an exhaustive two-way split grid in 10 % steps on Arbitrum, within 0.13 bps of the grid maximum on Base, and grid-exact on Robinhood Chain.

2026-11-16, `969bcc6`, fork job 133367255593 (`FOUNDRY_PROFILE=default`). The `FidelityMatrix` prints 12 lines of "fidelidade bps", and all 12 are 0 bps (Arbitrum 2, Base 6, Optimism 2, Robinhood Chain 2). One further Base line printed no execution ("exec real 0"), so it is not counted among the 12. The gap between the gross quote and what is delivered, printed alongside, is 27 to 28 bps on every line. Log lines 1231 to 1414 of that job.

CHAIN	EXECUTED LINES AT 0 BPS	SOLVER AGAINST THE GRID MAXIMUM (`969BCC6`)
Arbitrum	2	405150655106280137 against 405150948354222448: −0.0072 bps
Base	6	533611958127074266 against 533549116344743073: +1.178 bps, the Solver is above the grid
Optimism	2	equal to the wei (0 bps)
Robinhood Chain	2	equal to the wei (0 bps)

The grid comparison is `(s - g) / g × 10⁴` with `s` the Solver's output and `g` the grid maximum, read from the "[GRELHA] … solver/gridMax" lines. A Solver result above the grid maximum on Base means the grid, which is a 10 % grid, is not the optimum there; it is not a claim that the Solver is better than the optimum.

The 2.3 paper also stated real fills of "222,000 to 326,000 gas all-in" on Base. This paper does not repeat that as confirmed: the 2026-11-16 matrix prints "gas: est / exec real" and the exec figure runs 383,452 to 2,164,869 on the six executed Base lines (and the same interval on the 12 lines of the four chains). That the "exec real" column and the 2.3's "all-in" are the same quantity was not checked, so the two ranges are shown, not reconciled.

The matrix executes constant-product, concentrated, V4 (native included) and Solidly legs end to end. 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.

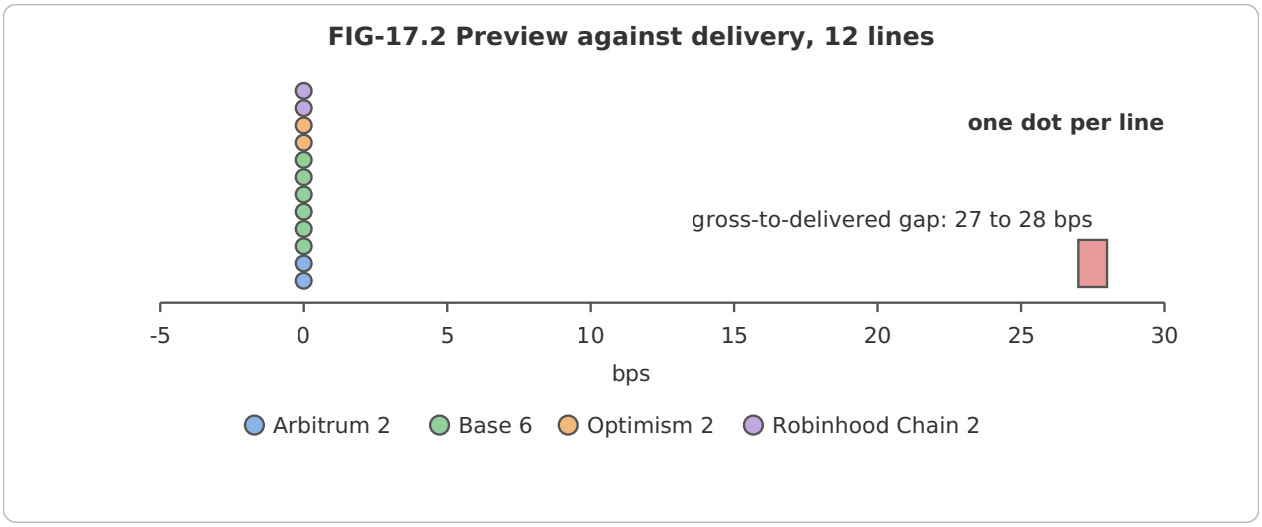

FIG. 17.2 Preview against delivery, 12 lines

FIG-17.2: preview against delivery, 12 lines, one dot per line, all at 0 bps, with the 27 to 28 bps gross-to-delivered gap drawn separately.

§17.3 The factory census and the public sweep

A venue registration is a claim, and the claim has its own instrument. The census harness (`FactoryCensus`) inherits the production deployment wiring and lives with that script in the private deployment tree, so its numbers are reported, not reproducible from this checkout. 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), over one denominator, hop-pairs, with an explicit unresolved bucket that measured zero on all four chains in 2026-08.

ARBITRUM, 88 HOP-PAIRS (2026-08, PRIVATE HARNESS)	HOLDS A POOL	WON THE PAIR
Uniswap V4 (native + wrapped)	67	48
Uniswap V3	86	33

ARBITRUM, 88 HOP-PAIRS (2026-08, PRIVATE HARNESS)	HOLDS A POOL	WON THE PAIR
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. There is no newer census in the repository. What is public and current is the sweep that shares its fixtures and its anti-vacuity gate, `test/fork/TokenSweep.t.sol`: in the fork job of 2026-11-16 (job 133367255593) it ran on Arbitrum (3 tests), Base (15) and Optimism (3), all green. A sweep is a pass or a fail on a fixed corpus; the paper does not turn it into a coverage percentage. The anti-vacuity gate means a pair reported as having depth must also quote non-zero.

§17.4 Gas

Every figure is traceable to a named harness or log, and cold and warm are reported separately. The table keeps the 2.3 rows with their date and adds the 2026-11-16 rows beside them. The two sets were taken with different set-ups and are not subtracted from one another.

MEASUREMENT	2.3 (2026-08)	2026-11-16 AT `969BCC6`
discovery on its own (`discoverFor`)	94,566	140,069
cold solve, first touch (`findBestRoutePlan`)	169,093	506,007 (COLD, first touch); 506,094 (COLD, hot slots)
warm solve	132,691	curated test: 433,610
warm saving	−36,402 gas (−21 %)	−72,340 gas (−14.30 %), 433,610 against 505,950
after the TTL	not measured	curated test: 564,239; after swaps (tier-2 lines) 564,253; after TTL 564,614
marginal cost per additional leg	about 28,700	73,955 (V2: `one_leg_tx` 200,835, `two_legs_tx` 274,790)
marginal cost per additional hop	via-bridge second hop about 59,000	125,666 (`one_hop_v2_tx` 200,880, `two_hops_tx` 326,546)

Sources. The 2.3 cold and warm rows come from `REPORTS.md:301-311` (offline LifecycleMetrics, 4 factories, 2026-08-04); the 2.3 cited `test/fork/DiscoveryColdWarm.t.sol` for them, and that file prints legs, gross output and pool per phase, with no gas, at `969bcc6` (log 133367255593:1156-1171). The November rows come from the `LifecycleMetrics` of job 133367307807 (`isolate=true`, default profile, log lines 7758 to 7810; the curated saving is `433,610 - 505,950`), and the marginal rows from the committed snapshots `snapshots/test_Gas_Leg_V2_OneAndTwoLegs.json` and `test_Gas_Leg_TwoHops_V2ThenV3.json`, checked by the gas snapshot guard (job 133367308176, `FOUNDRY_PROFILE=release`, `FORGE_SNAPSHOT_CHECK=true`; the snapshots as read at `969bcc6`, where they last changed at `4d5ee34`, 2026-11-14; at `6387350` they read 73,252 per leg (200,835 becomes 200,328; 274,790 becomes 273,580) and 125,481 per hop (200,368; 325,849)). That the November marginal rows (whole transaction) and the 2.3 rows (execution) measure the same quantity was

not checked; likewise that the November set-up matches the August one. The 2.3's `previewPlan` on live Base (1,524,221 / 1,414,633 / 1,359,925 gas) has no newer measurement: `test/fork/QuoterGasFork.t.sol` exists at `969bcc6` but does not run in the fork job, so those three numbers are carried as 2026-09 documents give them and are not re-measured.

The gas campaign G7/G8. `docs/GAS_CAMPAIGN_G7_G8.md` records a campaign of 5,600 result lines from CI run 43376232767 (sealed at `e21e8f5`, 2026-11-13; seven chains, 800 lines each, Avalanche with no usable line), re-measured locally at `bb4cdbb` on 2026-11-14 (the other six chains, 800 lines each, Avalanche absent). The figures below are the campaign's, not of `969bcc6`: they are medians (p50) of three recorded columns for the "major" kind in mode D, with the number of lines `n`.

CHAIN	P50 G_QUOTE	P50 G_RECORD	P50 G_EXEC	N
Base	11,759,662	686,452	1,153,096	20
Optimism	3,622,412	292,858	607,702	28
Arbitrum	5,127,965	356,052	719,544	30
BNB Chain	3,075,085	320,455	690,914	22
Ethereum	1,694,769	415,570	774,427	30
Robinhood Chain	1,350,754	327,136	1,127,205	9 (p95 = the maximum)
Avalanche	0 lines ok (770 no-quote, 30 unforced)			not measured

Avalanche is "not measured", not "zero cost": it produced no usable line. Robinhood Chain rests on nine lines, so its median is a small sample. The column definitions are the campaign document's (lines 19 and 53 to 69); this paper does not restate them.

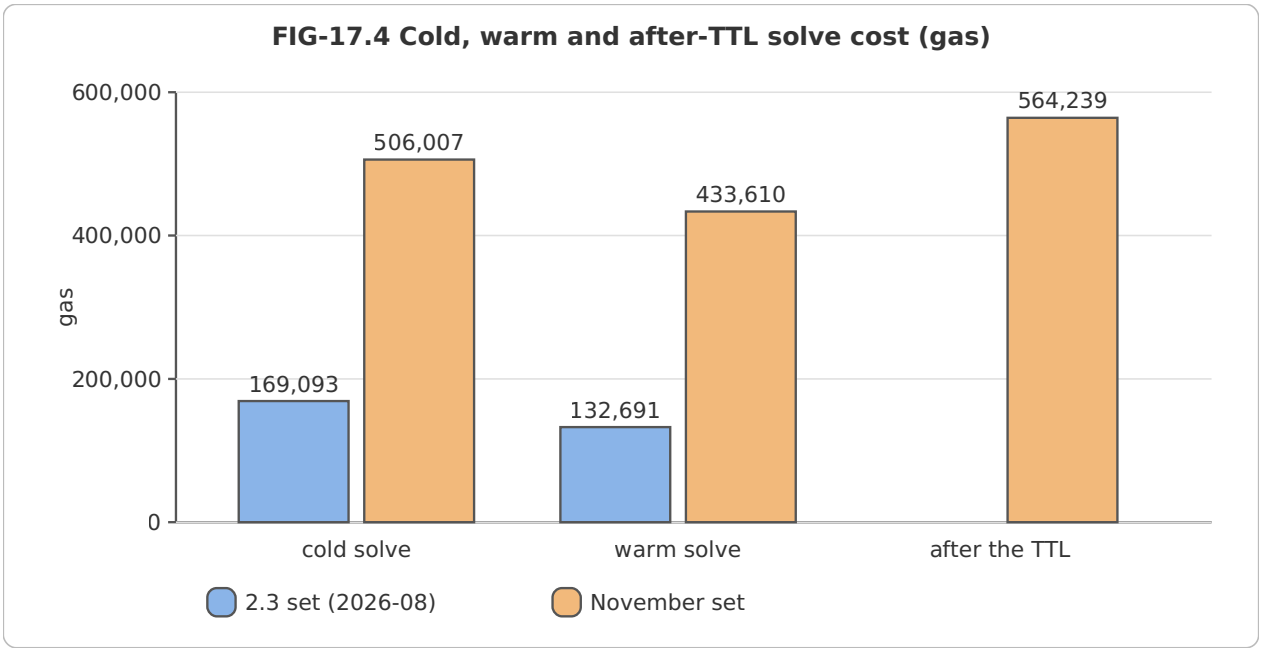

FIG. 17.4 Cold, warm and after-TTL solve cost (gas)

FIG-17.4: cold, warm and after-TTL bars from the table above, one colour for the 2.3 set and one for the November set.

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. A live quote is view computation of about 1.36 to 1.52 million gas (2.3, 2026-09), paid by no one on chain.

§17.5 The executor against the field

A gas table without an external reference is self-praise, so in 2026-08 the execution layer was benchmarked on the same fork, at the same block, with real production calldata of KyberSwap's `MetaAggregationRouterV2`, on a Base fork pinned to the block each route was built at (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, were run with our routes beside them, and both sides' deliveries were 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. Across nine measured pairs the fully solving door and the calldata door delivered identical output. The 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 % above its estimate.

The harness (`BenchExec.t.sol`) lives with the captured calldata in the private tree, so these numbers are reported, not reproducible from this checkout, and there is no newer measurement. The 2.3 added that the build was "compiled at `optimizer_runs = 999999`, so the difference is design, not compiler settings". That clause is withdrawn here: `foundry.toml` at `6387350` sets `optimizer_runs = 200` in both the default (line 19) and release (line 76) profiles. Whether the 2026-08 harness was compiled with another setting in the private tree is not known; until it is checked, this paper does not claim that compiler settings played no part. What can be said without it is the structural reading: their router interprets a generic call list that a server decided, ours executes a typed structure, and interpretation costs.

§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. 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.

2.3, 2026-08 (release profile). ρ* ran 66.9 gas per byte on single-leg routes to 105.5 on four- and five-leg routes, whole transaction, measured with `test/EquationBench.t.sol`.

2026-11-16, `969bcc6` (default profile, `isolate=true`, job 133367307807, log lines 11462 to 11520). `EquationBench` at this commit does not itself compute ρ*: it prints the gas and the calldata size of the two doors (`BENCH|gas` , `BENCH|cd_total`). The ratio Δgas/Δbytes between the solving door and the calldata door, computed here from those lines as `(bestExactIn - exactIn) / (cd_exactIn - 196)` , is:

CASE	ΔGAS	ΔBYTES	ΔGAS/ΔBYTES
S1	83,278	832	100.09
H2	149,068	1,376	108.33
S3	293,240	1,472	199.21
H2L4	407,735	2,016	202.25
S5	447,289	2,112	211.78

Whether this ratio is the same quantity as the 2.3's ρ* is inferred, not measured, and the two ranges (66.9 to 105.5; 100.09 to 211.78) are not the same measurement: the profile (release then, default now) and the isolation mode (`isolate=true` now) differ. They are shown together so a reader can see the order of magnitude, about one hundred to two hundred execution gas per calldata byte, and not a single number.

The byte prices of 2026-08 are carried as they were read then; no newer reading is in the repository (they appear in the repository only in `docs/papers/llms-full.txt` , a copy of the 2.3):

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 was 0.9 to 14 % of a swap's cost in 2026-08, so execution is where the money goes. The Router's four doors are not a convenience feature; they are the affine cost model, made into an interface. Byte prices move with the chains' fee markets, so the table is a dated reading, not a standing quote.

§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 core contracts, one mathematics. Three supporting pieces stand beside them: the Gate (with its per-venue wrappers), the Math library, and the adapters that translate an external venue into the Gate's one interface (§16.1).

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 others.
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.
Gate + adapters	Reach a venue whose arithmetic is not ours (Balancer V2 and V3, Curve NG) only through an immutable wrapper that is itself measured.

Six venue kinds, four families, and one door for the rest. The kinds the Core prices itself are Uniswap V2, V3 and V4 (native ETH included), Algebra dynamic-fee pools and Solidly stable and volatile pairs: six kinds with attributes, four working families, each priced by one closed form in the Core and reached by one dispatcher Ω (§4). A seventh kind, `EXTERNAL = 9`, is the Gate's: a venue that is not ours to price enters as a wrapper the Hub has registered, carries no attributes of its own, and is outside the executable set the Core itself dispatches (`BlazePhoenixCore.sol:273-277, 313-320` and `BlazePhoenixHub.sol:209-216` at `6387350`). The 2.3 said that Curve and Balancer were excised; that is no longer true, and it is not repeated here: Balancer V2 and V3 and Curve NG enter through the Gate, under the bounds of §14.4 (a delay on a new wrapper and a cap on the number of wrappers per pair), and the retired kind numbers stay retired (§4.1). Which venues are claimed is Appendix E, and only those the repository's README lists.

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. A venue's own account of what it delivered. 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. The site's deployment record lists V1 (version 1.0.0, "live") on five networks: Ethereum, Base, Optimism, Arbitrum and Robinhood Chain (`/api/deployments` and `/api/manifest`, read 2026-11-16). The 2.3 measured the deployed V1 router by RPC on 2026-09-04: the same 14,624-byte contract on Base, Ethereum, Optimism and Arbitrum (Robinhood Chain was not measured then, and this paper has not repeated the measurement). `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 is 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 described in this paper builds a 22,717-byte Router (23,781 in the 2.3) 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 `6387350` (measurements keep the commit they were taken at) (the 2.3 stood at `8949a9d`). The site's record for V2 reads "pending" with no address on any network, and the Gate's adapter record in the repository has a header and no rows: no adapter is deployed.

V2 launches 15 December 2026, on six target networks. The targets are Base, Optimism, Arbitrum, Robinhood Chain, Ethereum L1 and Avalanche. They are targets and not deployments: no address is given for any of them, here or in any list in this paper, and the launch date is a schedule, not a record. The repository's README at `6387350` names the same six targets (README.md:31); the README at `969bcc6` named Base, Optimism, Arbitrum, Robinhood Chain and further L2s and said V2 is not planned for Ethereum L1. On Avalanche the gas campaign produced no usable measurement (§17.4). The README gives no launch date: the date is the owner's current statement, and the reader who wants the live truth reads the deployment record, not either document.

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, in 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. The staking engine follows the same rule: its two deployments are in §25.8, and it too has no address until it is deployed.

§27 Interfaces: the SDK, the MCP servers, the API and the Hermes plugin

Plain: six ways to reach the DEX, and none of them holds your keys or your funds: a library, two tool servers for AI agents, an HTTP API, an agent plugin and the APP. On the five read from source, the quote is always an `eth_call` on a node you choose; the APP is still a design document (§27.6). *Precise:* each interface is a client of the same on-chain Quoter and Router; none adds an authority, none signs, and each of the five read from source carries the Phoenix Check and a field-by-field comparison of the swap calldata with the request, computed where the reader can re-run it. This chapter says, per surface, which commit was read, who performs the RPC, what is checked and what is not guaranteed. Every statement about behaviour cites a source line as `path:line@commit`; the four interface repositories are each under the MIT licence (their `LICENSE` files), which is not the licence of the contracts.

§27.1 The common model

Three properties hold on every surface read from source (the APP, a design document, is outside them), and each is a property of the source, not of a deployment.

No keys, no funds. No interface receives, stores or derives a private key. In the SDK source a search for `privateKey`, `mnemonic`, `signTransaction` and `sendRawTransaction` finds nothing; the only transaction send is `execute`, which requires a wallet client supplied by the caller and refuses to run without one (`src/client.ts:822-895@2d2fc39`). The local MCP server does not expose `execute` in any tool (`src/server.ts:10@3514804`), the hosted surfaces return unsigned calldata only, and the Hermes plugin states the same of itself (`bp.py:12@cf38638`).

The quote runs on the caller's node. No surface ships a default RPC endpoint. With no RPC configured the SDK throws `rpc_required` and does not fall back to a built-in endpoint (`src/rpc.ts:165-171@2d2fc39`); the Hermes plugin does the same (`bp.py:225-226@cf38638`); the hosted surfaces never run the call at all (§27.4).

The check is recomputed, not relayed. The Phoenix Check has one definition and three implementations that carry the same two thresholds, 2000 bps and 200 bps: the SDK (`src/checks.ts:9-10@2d2fc39`), the API worker (`worker/codec.ts:51-117@01c5468`) and the Hermes plugin (`bp.py:56@cf38638`). That the three agree on every input is not established here; only the thresholds and the verdict order were compared.

§27.2 The SDK

The package `@blazephoenix/sdk` is at version 1.6.0 on npm (latest, published 2026-11-20; the npm record names commit `bb41243`). The repository main, `2d2fc39`, carries 1.7.0, which is not on npm. It has two entry points, the package root and `@blazephoenix/sdk/xray`.

Client. The class `BlazePhoenix` takes the RPC from the caller: a URL, a list of URLs in fallback order, an EIP-1193 provider, a viem transport, or a per-chain map of these (`src/client.ts:49-55@2d2fc39`). RPC URLs must use `https://` or `wss://`; plain `http://` and `ws://` are accepted only for a local node (`src/rpc.ts:55-68@2d2fc39`). The client offers fourteen methods: `quote`, `usdValue`, `quoteExact`, `quoteBatch`, `buildSwap`, `simulate`, `execute`, `tokenInfo`, `solvency`, `getFills`, `watchFills`, `deployments`, `re-`

`solveDeployment` and `verifyDeployment`. The set is the same in 1.6.0 and on main (`src/client.ts:138-975@bb41243`, `src/client.ts:138-987@2d2fc39`). It covers five chains: Ethereum, Base, Optimism, Arbitrum and Robinhood Chain (`src/constants.ts:19@2d2fc39`).

Phoenix Check. `phoenixCheck` derives its result only from the Quoter's preview. The verdict is `blocked` when the preview cannot execute or the route is inconsistent; otherwise an expected impact of 2000 bps (20 %) or more is `danger`, 200 bps (2 %) or more is `caution`, and anything below is `ok` (`src/checks.ts:29-32@2d2fc39`). `routeIsConsistent` requires at least one hop, the first hop starting at tokenIn, the last ending at tokenOut, and each hop starting where the previous one ended (`src/checks.ts:19-25@2d2fc39`). One field deserves a plain reading: `ironFloor.enforcedOnChain` is a constant `true` in the returned object, a statement about the contract (§11.5), not a value the SDK reads from chain (`src/checks.ts:42-48@2d2fc39`).

Calldata verification. `verifySwapExactIn` decodes the bytes with the Router ABI and requires the selector of `swapExactIn`; it then checks the amount in, the recipient, the deadline, a minimum out that is non-zero and at least the requested floor, and that the route connects tokenIn to tokenOut. Any mismatch throws `calldata_mismatch` with every mismatching field listed; the function performs no I/O (`src/calldata.ts:52-98@2d2fc39`). Its reach is narrower than its name suggests: `buildSwap` calls it on one path only, the 2.x path where the Quoter returns ready calldata for an ERC-20 input in route mode (`src/client.ts:718-743@2d2fc39`). On the other paths (1.x, native input, or best mode) the SDK encodes the Router calldata itself from the preview, so there are no foreign bytes to verify (`src/client.ts:744-758@2d2fc39`).

Simulate and execute. `simulate` runs the swap as an `eth_call` from the paying address on the caller's RPC (`src/client.ts:804-820@2d2fc39`). `execute` sends through the caller's wallet, and rebuilds a plan older than 30000 ms when it is given the request, refusing to send a stale one otherwise (`src/client.ts:833-873@2d2fc39`).

USD value. `usdValue` prices an amount only from the Quoter's own net-out answers, after the protocol fee and including price impact, read in one pinned block; the dollar anchor is the chain's USDC taken as 1 USD (USDG on Robinhood Chain) (`src/usd.ts:1-11@2d2fc39`). It references no external price API and no oracle, and when nothing can be measured it answers `not measured`, never 0 (`src/usd.ts:214-220@2d2fc39`). A stablecoin taken as 1 USD is an assumption, and the result is a quoted price, not an executed one.

Cross-chain price (1.7.0, main only). `src/crossChain.ts` is new on main and absent from the npm package. `crossChainPrice` runs `usdValue` for one token on each listed chain, each on that chain's own RPC, and returns a per-chain price with its derivation: the route that gave it, leg by leg (`src/crossChain.ts:298-408@2d2fc39`). Tokens are named per chain by address or by a short built-in list, never matched across chains by on-chain symbol or name (`src/crossChain.ts:10-12@2d2fc39`). `aggregateQuotedPrices` is pure integer arithmetic over those answers (`src/crossChain.ts:112-149@2d2fc39`). Each price says whether its impact is `measured` or `modeled`, by venue kind (`src/crossChain.ts:345-383@2d2fc39`). The module bridges nothing and sends no transaction; its own basis string says the prices are quoted, not executed, and include the 0.28 % fee (`src/crossChain.ts:385-388@2d2fc39`).

X-Ray. The second entry point measures a token before a trade: can it be bought, sold and transferred at the tax the caller accepts. The caller must state `maxTaxBps`; the SDK does not pick it. It measures by simulation on a pinned block, on a ladder of tiers from `eth_simulateV1` down to a plain `eth_call` (`src/xray/tiers.ts:1-4@2d2fc39`), with I/O only through a transport the caller injects (`src/xray/index.ts:`

1-16@2d2fc39). Verdicts form a lattice, CONDEMN below WARN below MEASURED, with NOT_MEASURED as a status and not a pass (src/xray/lattice.ts:1-33@2d2fc39). It runs on three chains only (Base, Ethereum, Arbitrum), where it pins the Router and Quoter by address and code hash (src/xray/ctx.ts:32-39@2d2fc39), and its limits are declared in code per tier (src/xray/tiers.ts:425-436@2d2fc39). A MEASURED verdict describes one amount, one block and the measured actors; it says nothing about what a token's owner does next.

The one request that is not to the caller's node. In its default registry mode, `auto`, the SDK fetches the deployment record from the site (`/api/deployments`) before its first read (src/constants.ts:13-17@2d2fc39 , src/registry.ts:83-90@2d2fc39). The fetched record cannot move an address the package embeds, a deployment known only from the fetched record is verified on the caller's RPC before first use and refused on failure, and if the fetch fails the embedded snapshot is used (src/registry.ts:5-14@2d2fc39). Registry mode `embedded` makes no such request. No telemetry or analytics code was found. `verifyDeployment` checks that each registered address has code and, on 2.x, that `VERSION()` and the hub and solver wiring match the record; it compares no code hash (src/client.ts:191-243@2d2fc39).

§27.3 The local MCP server

The package `@blazephoenix/mcp` is at version 1.3.0 on npm (latest, published 2026-11-20; repository 3514804), binary `blazephoenix-mcp`, depending on the SDK at exactly `1.6.0` (package.json:26-31@3514804). It is a thin layer: it runs on the user's machine and every chain read goes through the SDK client, so §27.2 applies to it as 1.6.0 stands, including the default registry fetch, which no environment variable of the server changes (src/server.ts:128@3514804).

It registers eight tools (src/server.ts:134-293@3514804): `get_quote`, `build_swap`, `simulate_swap`, `check_solvency`, `get_token_info`, `get_deployments`, `verify_deployment` and `xray_token`. `build_swap` returns the SDK's plan, unsigned; `simulate_swap` requires the paying address and returns a reverted simulation as an error (src/server.ts:181-207@3514804). `check_solvency` reads no contract and answers `coming_soon`, an error result and never a verdict (src/server.ts:209-218@3514804); the reading of §25.10 for the staking engine is the SDK's `solvency` method. `xray_token` carries the SDK X-Ray's requests to the configured node and shows MEASURED only when the SDK verdict and every instrument are MEASURED, so the server's own predicate can lower a verdict and cannot raise one (src/xray.ts:62-70@3514804). Every input schema is strict, so an unknown key is refused and not dropped (src/server.ts:34-36@3514804).

The RPC comes from the environment, never from a tool argument: the server hands it to the SDK's `rpcFromEnv` (src/server.ts:128@3514804), which reads `BLAZEPHOENIX_RPC_<CHAIN>` per chain (by name or by chain id) or `BLAZEPHOENIX_RPC_URL` for a single node, and refuses both at once (src/rpc.ts:217-236@bb41243); the start-up log names the chains configured, never the URLs (src/server.ts:75-87@3514804). The protocol version it serves is chosen per call or pinned by `BLAZEPHOENIX_VERSION` (`latest` , a major such as `1` or `2` , or a full version such as `2.0.0`); any other value stops the server at start (src/server.ts:124-131@3514804). That is how V1 and V2 stay separate.

§27.4 The hosted MCP and the API

The API is a worker with one codec and two faces: REST routes and a hosted MCP endpoint. Its repository was read at `01c5468`. Its design is that the server prepares and decodes, and the caller's node answers.

Prepare, call, decode. `prepare` returns the exact `eth_call` as a JSON-RPC object; the caller runs it on their own node and posts back the request with the node's result; `decode` turns that into the quote, the Phoenix Check and calldata (`worker/codec.ts:235@01c5468`). The plain quote routes behave the same way: asked for a quote without a result, they answer `rpc_required` together with the prepared call (`worker/quoteApi.ts:615-623@01c5468`, `worker/quoteApi.ts:713-729@01c5468`). The hosted MCP serves five tools over JSON-RPC 2.0 at `POST /mcp`, without a session (`worker/mcp.ts:1-2@01c5468`): `prepare_quote`, `decode_quote`, `decode_usd_value`, `get_deployments` and `get_abi` (`worker/mcp.ts:72-177@01c5468`). `prepare_quote` calls the same function as the REST prepare route (`worker/mcp.ts:199-211@01c5468`).

What the server does not do. On these paths the worker makes no outbound request: a search for `fetch(` finds none in `worker/mcp.ts`, `worker/codec.ts` and `worker/usd.ts`. This is narrower than "the worker performs no RPC". The same worker serves site routes (`/api/stats`, `/api/tape`, `/api/verify`) that do read chains from the operator's side; those reads feed no quote served to a caller. A quote function that reads through the operator's nodes for the site's own bot exists in `worker/quoteApi.ts` and is not routed from any public path (`worker/quoteApi.ts:628-632@01c5468`). The statement of this section is about the quote, USD and MCP paths only.

Fail closed. `decode` re-validates the posted request field by field and refuses an unknown chain or version. A result that does not decode as the Quoter's answer is `decode_failed`; a preview that is not consistent with the request is `result_mismatch`; an empty answer is `no_route` (`worker/codec.ts:281-339@01c5468`, `worker/codec.ts:461-465@01c5468`, `worker/codec.ts:632-634@01c5468`). When a recipient is given, the Quoter-returned calldata is checked field by field (function, amount, recipient, deadline, minimum, route) before any transaction is offered (`worker/codec.ts:542-553@01c5468`). And without a `userMinOut` set by the caller, `decode` offers no calldata at all (`min_required`): the code's own reason is that a pure decode cannot tell the node's answer from a forged but self-consistent one (`worker/codec.ts:527-535@01c5468`).

What binds the answer to the request, and what does not. There is no request hash linking `prepare` to `decode`: `decode` receives the request object as sent and re-parses it (`worker/codec.ts:281-283@01c5468`). The quote's `eth_call` is prepared at block tag `latest`, and the decoded quote carries no block number (`worker/codec.ts:235@01c5468`). Only the USD value pins a block, by block hash, and rejects a second-stage result from a different block (`worker/usd.ts:66-109@01c5468`, `worker/usd.ts:284-302@01c5468`). So a quote decoded here is as fresh and as honest as the node that answered and the moment it answered; the floor the caller set is what the contract enforces (§11.5), and simulating the calldata from the paying address before sending remains the caller's step.

Solvency. The solvency page and badge read no contract and say that proof of solvency is coming soon (`worker/solvency.ts:1-33@01c5468`).

§27.5 The Hermes plugin

The Hermes plugin is a plugin for the Hermes agent runtime: its manifest declares version 1.0.0 and requires Hermes 0.23.0 or later, and it gives an agent four tools: `blazephoenix_quote`, `blazephoenix_verify_calldata`, `blazephoenix_usd_value` and `blazephoenix_chains` (`plugin.yaml:`

`2-14@cf38638`). It is a single Python module on the standard library, with the Router and Quoter addresses compiled in per chain (`bp.py:29@cf38638`) and its own implementations of the route check, the Phoenix Check and the calldata verification (`bp.py:270-284@cf38638`).

The RPC URL is read from the agent's secret store under `BLAZEPHOENIX_RPC_<CHAIN>`; without it the tool answers `rpc_required` and names the variable to set (`bp.py:222-226@cf38638`). The module has one network call site (`bp.py:214@cf38638`). The JSON-RPC methods it sends to the node are `eth_call`, `eth_chainId` and `eth_getBlockByNumber`, reads only (`bp.py:351@cf38638`, `bp.py:373@cf38638`, `bp.py:477@cf38638`, `bp.py:487@cf38638`, `bp.py:645@cf38638`).

The quote tool is not on the node alone. It quotes protocol version 1, and it asks the hosted API (§27.4) to prepare the call and to decode the node's answer, so the request, and the node's result, are sent to the site (`bp.py:335-339@cf38638`, `bp.py:354@cf38638`). It does not take those answers on trust: the prepared call must match the Quoter and Router compiled in and the request, or the tool refuses with `prepare_mismatch` (`bp.py:345-350@cf38638`); the eth_call itself runs on the user's node (`bp.py:351@cf38638`); the hosted decode must agree with the plugin's own decoding of the node's result, or it refuses with `decode_mismatch` (`bp.py:355-359@cf38638`); the Phoenix Check is computed locally and reported beside the hosted verdict with a flag saying whether they agree (`bp.py:379-386@cf38638`); and any calldata the hosted decode returns is checked by the plugin's own `verify_swap` before it is shown, unsigned (`bp.py:393-399@cf38638`). That check compares the target with the chain's Router, requires a zero `value` and the `swapExactIn` selector, decodes the calldata and compares `amountIn`, the recipient, `userMinOut` (at least the requested minimum, and never zero) and the deadline (equal to the request's and within the next 3600 seconds); it requires the route to connect the input token to the output token and to equal the route the node returned, and, when an approval is returned, that it is exactly `amountIn` of the input token to the Router. Any difference refuses with `calldata_mismatch` and the list of mismatches (`bp.py:284-328@cf38638`). The USD value tool and the chain list make no request to the site.

One property is specific to an agent setting. An RPC URL often carries its key in the path or the query, and a node's error text can echo that URL back, which would hand the key to the model and to its transcript. Since `cf38638`, `_scrub` removes from the node's text every piece of the URL that can carry a key (path segments raw and decoded, query values, user info, host), replaces each with `[redacted]`, and reduces any URL left to its scheme; where a credential is too short to remove safely, the node's text is dropped altogether (`bp.py:233-254@cf38638`). `rpc` passes every node error through it before the agent sees it (`bp.py:257-262@cf38638`).

§27.6 The APP

The APP. The site's APP is a separate route (`/app/`) from the content site. By its design (DESIGN-APP.md §2) it has no account, no email, no backend of its own, no wallet analytics and no custody; it adds no trust to the trade and sits in the same hostile-front-end threat model as any front end (§14.5). Its swap panel keeps the minimum received visible at all times (the floor the contract enforces, §11.5), shows the age of the quote in seconds, and prints a receipt with "Quoted" beside "Received", both measured (the stored quote and the balance delta of the receipt); if they differ, the difference is shown as it is. A refusal is shown with its own message when the error selector is known and as the selector in hexadecimal when it is not, and "nothing left your wallet" is shown only when the receipt proves it. The design document names neither a Phoenix Check

nor the node the APP reads from, so this paper claims neither for the APP. Whether the APP is live at the time of reading is not established by this paper: DESIGN-APP.md is a design document, and the paper makes no claim about its deployment state.

§27.7 Summary, and what was not checked

SURFACE	VERSION, COMMIT	WHO PERFORMS THE RPC	WHAT IT VERIFIES	WHAT IT DOES NOT GUARANTEE
SDK	1.6.0 on npm (`bb41243`); 1.7.0 on main (`2d2fc39`)	The caller's node; one registry fetch to the site in `auto` mode	Phoenix Check; calldata against the request on the 2.x route path; remote deployments before first use	That the node is honest; that a quoted price is an executed one; code hashes in `verifyDeployment`
Local MCP	1.3.0 on npm (`3514804`), SDK 1.6.0	The user's node, from the environment	What SDK 1.6.0 verifies; X-Ray verdict can only be lowered	Solvency (`coming_soon`); anything the SDK does not
Hosted MCP and API	`01c5468`	The caller's node; the server runs no call on the quote paths	Request re-validated; result decoded and checked for consistency; calldata field by field; no calldata without `userMinOut`	The block of the answer (`latest`, unpinned); a binding hash between prepare and decode; the node's honesty
Hermes plugin	1.0.0 (`cf38638`)	The user's node, from the agent's secret store; the quote's prepare and decode steps go to the hosted API	Hosted prepare and decode checked against the pinned addresses and the node's result; route check, Phoenix Check and calldata verification in its own code; node error text scrubbed	That the request stays private from the site on the quote tool; agreement with the SDK on every input (not compared here)
APP	design document	Not stated in the design document	By design: floor shown at all times; quoted beside received	That it is live (§27.6)

TAB-27.1: the interfaces, by who performs the RPC and what each verifies.

What this chapter did not establish. The test suites of the four repositories were not run for this paper, so no pass count is given. The hosted endpoints were not probed: the chapter describes the repository at `01c5468`, not what is served at the time of reading. The npm packages were matched to commits by the commit the npm record names, not by comparing the published files. The Hermes plugin was read by targeted search and through its quote path (`bp.py:333-402@cf38638`), not line by line. The three implementations of the Phoenix Check were compared by thresholds and verdict order only. And on every surface the final defence is the same and is not the interface: the minimum out the caller sets, enforced by the Router (§11.5).

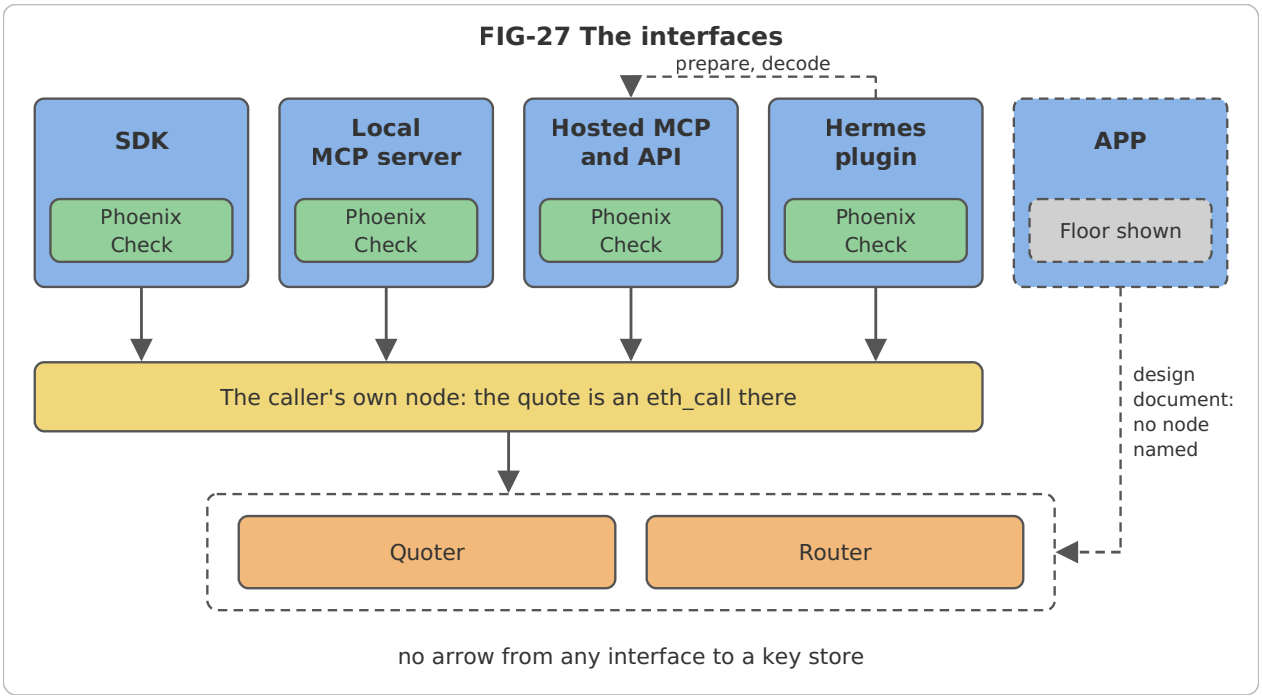

FIG. 27 The interfaces

FIG-27: the interfaces as boxes in front of one Quoter and one Router: the SDK, the local MCP server, the hosted MCP and API, the Hermes plugin and the APP. The quote arrow of each of the first four passes through the caller's node and the Phoenix Check sits inside each of them; a dashed arrow marks the Hermes plugin's prepare and decode requests to the hosted API. The APP, a design document (§27.6), is drawn dashed: it shows the floor, and the design names no Phoenix Check and no node. No box has an arrow to a key store.

Part VI — Staking V4

§25 Staking V4: Design

Staking V4 is the current codebase; the address is published at launch. It is not deployed yet, and no address is official until the deployment is published.

This chapter describes the staking engine as its source stands: `src/BlazePhoenixStaking.sol` in the `Blaze-Phoenix-Staking` repository, `main` at `9958d5f`, which declares `VERSION = "4.0.0"` (L177). Every line number in this chapter (L*n*) refers to that file at that commit; where this text and the source disagree, the source wins. The chapter carries no addresses, no deployment dates and no yields: every number in it is a constant compiled into the contract, or is derived from one, and the derivation is printed beside it. The repository states that the staking contract is a pre-launch engineering preview and that no external audit has been done: No external audit yet. Inspect it yourself: the code, the tests, the mutants, the bounty.

The engine is held to the discipline of the rest of this paper. It holds one token, BZPX, and every claim on that BZPX is a ledger term whose sum the contract re-checks around each state change (E35). A staker locks BZPX for a chosen number of days and earns two streams, a scheduled emission (E27–E29) and the interest borrowers pay (E31–E34), each weighted by a boost that grows with the commitment (E30). A staker may

borrow BZPX against the stake (E41). When a loss occurs, it follows one fixed order (E37), and the contract reports, as a public view, whether it holds what it owes (E36). The contract is deployed twice, independently (§25.8), and anyone may trip its circuit breaker on a proven shortfall (§25.11).

§25.1 Architecture and roles

One contract, `BlazePhoenixStaking`, inheriting `AccessControl`, `Pausable` and `ReentrancyGuard` (L175). The token, the treasury and the emission window (`emissionStart`, `emissionEnd`) are immutables (L286–289); the constructor refuses a token address that carries no code (L525). There is no oracle, no upgrade path and no parameter setter: every constant below is compiled in.

ROLE	HELD BY	MAY	LINE
`DEFAULT_ADMIN_ROLE`	the deployer, at construction	grant and revoke roles (no `_setRoleAdmin` call exists, so it administers every role)	L533
`ADMIN_ROLE`	the deployer, at construction	`fundEmission` (cumulative funding ≤ `TOTAL_REWARDS`); `withdrawReserve` (to the immutable treasury only, at most `protocolReserve`, closed under a hard breach); `sweepUndistributedEmission` (only after `emissionEnd`); `unpause`; `cancelEmergency` (only on a solvent book)	L534, L564, L582, L619, L642, L684
`GUARDIAN_ROLE`	nobody at construction; granted by a role admin	`pause`; `declareEmergency` — neither moves funds	L179, L640, L668
anyone	—	`liquidate`; `tripBreaker` (reverts unless the book is in hard breach); `absorbBadDebt`; `syncEquity` (in an emergency only; writes off a position whose stake no longer exceeds its debt); `lockStep`	L1211, L660, L711, L733, L1477

No role can move a staker's principal, and the only destination any role can send revenue to is the immutable treasury.

CONSTANT	VALUE	MEANING	LINE
`TOTAL_REWARDS`	90,000,000e18	emission budget of this contract, and the cap on cumulative `fundEmission`	L207
`BASE_REWARDS`, `HALVING_SERIES`	60,000,000e18, 32,000,000e18	the flat base (7,500,000 a year) and the geometric series that carries the 30,000,000 bonus (§25.3)	L208–209
`HALVING_PERIOD`	2 × 365 days	length of one emission period	L210
`EMISSION_PERIODS`	4	periods before the programme closes (`EMISSION_LENGTH` = 8 years)	L211–212

CONSTANT	VALUE	MEANING	LINE
`INITIAL_REWARD_PER_SEC`	`BASE_REWARD_PER_SEC` + `BONUS_REWARD_PER_SEC`	period-0 rate, $R_b + R_0$: base rate plus period-0 bonus rate (§25.3)	L213–216
`MAX_STAKE_PER_WALLET`	30,000,000e18	stake ceiling per position	L218
`MIN_EMISSION_WEIGHT`	1,000e18	below this total weight both streams are throttled pro rata	L228
`MAX_LTV`	50	borrow ceiling, % of net stake	L230
`LIQ_THRESHOLD`	95	liquidatable at debt ≥ 95 % of stake	L231
`LIQ_BONUS_BPS`	500	liquidation surplus to the caller, bps of debt	L232
`RESERVE_FACTOR_BPS`	300	share of each interest slice kept in `protocolReserve`	L233
`MIN_LOCK_DAYS`, `MAX_LOCK_DAYS`	90, 2,555	lock bounds in days	L241–242
`BOOST_BASE`, `BOOST_LINEAR`, `BOOST_QUAD`	10,000, 750, 250	boost polynomial, bps	L244–246
`MIN_DEPOSIT_BLOCKS`	10	blocks a deposit must age before withdraw, pure-yield claim or lock change	L248
`RATE_R0`, `RATE_RK`, `RATE_UK`, `RATE_S1`, `RATE_S2`	100, 500, 0.8e18, 500, 72,500	kinked interest curve, annual bps	L251–255
`MAX_PROTOCOL_UTIL_WAD`	0.75e18	book utilisation ceiling for a new borrow	L262
`CONSERVATION_DUST`	1e10	tolerance of the conservation guard, in wei	L267
`MAINT_BASE`, `MAINT_DENSITY`, `MAINT_GAP_UNIT`, `MAINT_MAX_SCAN`, `MAINT_MAX_LOCK_ACTIONS`	1, 50, 15 minutes, 10, 4	autonomous-maintenance budget (E42)	L272–275, L283

§25.2 The flow: stake, unstake, claim, borrow

Every user entry point runs one pipeline, in the order `deposit` fixes: snapshot the conservation identity; refuse if the book is in hard breach; advance the interest index and the emission window (`_updateGlobal`, L1620); settle both streams at the weight the position held; attribute the interest the position owes (`_accrueInterestFor`, L1905); clear an elapsed lock; perform the action; recompute the position's weights through the single writer (`_resync` → `_applyBoost` → `_checkpoint`, L2020–2062); run the autonomous maintenance window (E42); and check the conservation identity again (E35). The doors differ only in the gates they add.

ENTRY POINT	PAUSED	EMER-GENCY	HARD BREACH	FURTHER CONDITIONS
`deposit(amount, lockDays)` — stake	closed	closed	closed (L898)	$90 \le$ `lockDays` $\le \min(2{,}555,$ days left to `emissionEnd`$)$; position stake ≤ `MAX_STAKE_PER_WALLET`; before `emissionEnd` (L884–898)
`lock(lockDays)`	closed	closed	—	`MIN_DEPOSIT_BLOCKS` since the last deposit (L1107–1112)
`withdraw(amount)` — unstake	open	closed	closed (L1040)	no debt; lock elapsed; `MIN_DEPOSIT_BLOCKS` since the last deposit (L1027–1032)
`claimRewards()`	open	closed	closed (L1069)	settles emission and pure yield (L1067)
`claimPureYield()`	open	closed	closed (L1084)	no debt; `MIN_DEPOSIT_BLOCKS` since the last deposit (L1080–1084)
`borrow(amount)`	closed	closed	closed (L968)	E41 (L960–980)
`repay(amount)`	open	closed	open	— (L996)
`liquidate(user)`	closed	closed	open	E39 (L1211)
`emergencyWithdraw()` — exit	—	only	—	E40 (L772)
`absorbBadDebt(amount)`	open	open	open	E38 (L711)

Stake. `deposit` pulls BZPX into the position and starts or extends its lock; the lock may never run past the end of the emission programme, and the boost it buys (E30) is paid only while the lock is in force. **Unstake.** `withdraw` returns principal once the position owes nothing and its lock has elapsed; it stays open under a guardian pause, so a pause halts new risk without trapping anyone. In an emergency the only exit is `emergencyWithdraw`, which pays net equity, scaled pro rata if the contract holds less than the positions are owed (E40), and forfeits the position's unclaimed streams. **Claim.** The harvest is the two claims: `claimRewards` pays both streams, `claimPureYield` the same for a debt-free position; there is no separate harvest function. **Borrow, repay, liquidate.** A staker may borrow BZPX against its own stake within two ceilings (E41); interest is charged against the stake, not added to the debt (E34); `repay` stays open under a pause and under a hard breach, because closing debt only makes the book safer. A position whose debt reaches 95 % of its stake can be closed by anyone, and the closer receives the surplus of E39.

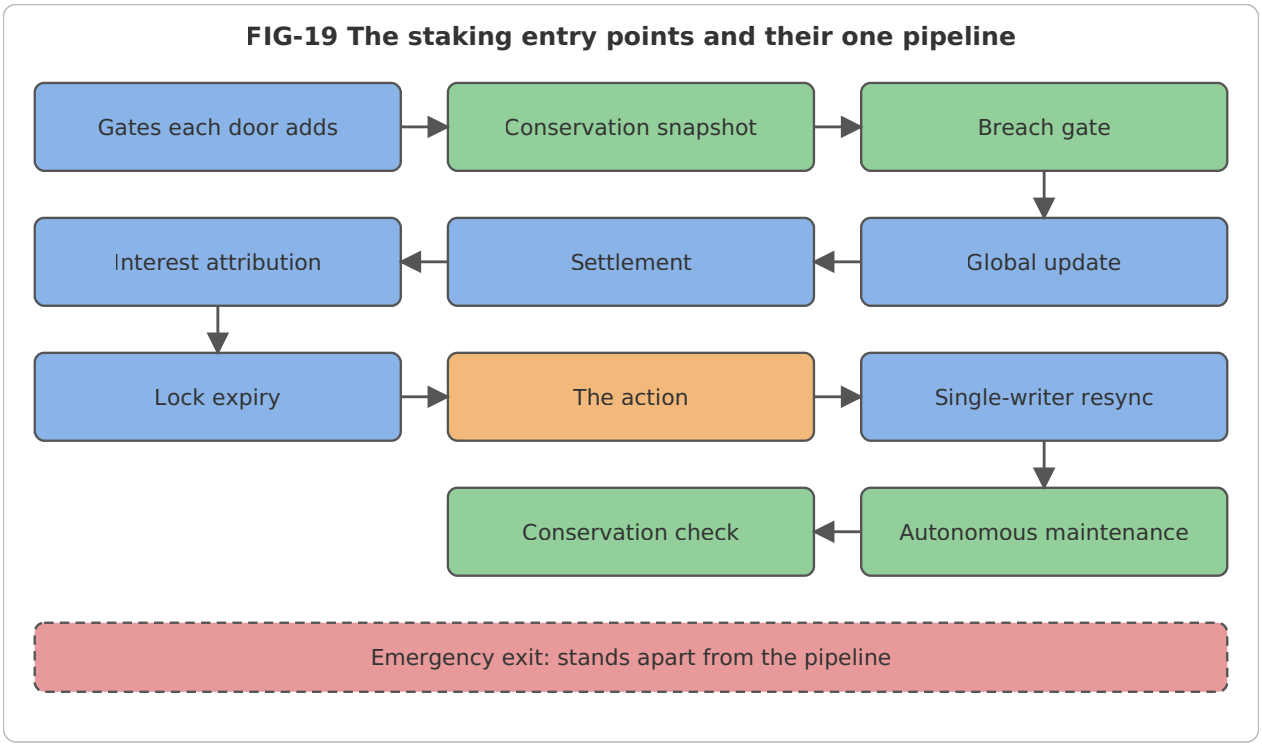

FIG. 19 The staking entry points and their one pipeline

FIG-19 — The staking entry points and their one pipeline: the gates each door adds, then the conservation snapshot, the breach gate, the global update, settlement, interest attribution, lock expiry, the action, the single-writer resync, autonomous maintenance and the conservation check; the emergency exit stands apart.

§25.3 Reward accounting

Emission. Each deployment schedules 90,000,000 BZPX over eight years. The schedule has two parts that add up to `TOTAL_REWARDS`: a linear base of 60,000,000 BZPX (`BASE_REWARDS`, L208), paid at a constant rate for the whole eight years, and a bonus series of 30,000,000 BZPX whose rate halves every two years (`HALVING_SERIES` = 32,000,000 BZPX, L209, of which the series contributes 30,000,000 BZPX, because its last step leaves 2,000,000 over: $S - S/16$). Let T = `TOTAL_REWARDS`, B = `BASE_REWARDS`, S = `HALVING_SERIES`, P = `HALVING_PERIOD` (2 × 365 days = 63,072,000 s), L = `EMISSION_LENGTH` = 4P = 252,288,000 s (L212), $R_b = \lfloor B/L \rfloor$ (L213), $R_0 = \lfloor (S/2)/P \rfloor$ (L214), s = `emissionStart`, and for a time t let $\tau = t - s$ and $p = \lfloor \tau / P \rfloor$. The cumulative scheduled emission is a closed form (`_emittedAt`, L1665–1675):

$$E(t) \;=\; \begin{cases} 0 & t \le s \\ R_b\,\tau \;+\; (S - (S \gg p)) \;+\; (R_0 \gg p)\,(\tau - pP) & s < t < s + 4P \\ B + S - (S \gg 4) & t \ge s + 4P \end{cases}$$

(E27)

Derivation. In exact arithmetic the bonus rate in period q is $S/(2^{q+1}P)$, so a full period q emits $S/2^{q+1}$ and the bonus emitted before period p is $\sum_{q<p} S/2^{q+1} = S(1 - 2^{-p})$. $S = 3.2 \times 10^{25}$ is divisible by 2^4, so $S \gg p = S/2^p$ exactly for $p \le 4$ (precondition `S % 2^4 == 0`) and the second term is that sum with no rounding. The two rates are floored: R_b is short of B/L and $R_0 \gg p$ is short of $S/(2^{p+1}P)$, each by less than one wei per second. Inside a period E therefore grows linearly and stays below the exact curve; at each rollover it steps up to the exact series, by 2,176,000 / 1,088,000 / 544,000 wei at the turns p = 1, 2, 3 and by 229,184,000 wei at the close (197,376,000 of base plus 31,808,000 of bonus) (`python` on the closed form). The

curve is monotone non-decreasing and it closes exactly:
$B + S - S/16 = 60{,}000{,}000 + 32{,}000{,}000 - 2{,}000{,}000 = 90{,}000{,}000\text{e}18 = T$, remainder 0 wei. Nothing is left unscheduled by the formula; an unfunded or empty window is a different matter: whatever the schedule did not pay out is recoverable to the treasury only by `sweepUndistributedEmission` after the close (L619). Per-year figures follow from the constants and are printed in TAB-25.3 so a reader can check each row by hand. The base contributes 7,500,000 BZPX each year (60,000,000 / 8). The bonus contributes 8,000,000, 4,000,000, 2,000,000 and 1,000,000 in the four two-year periods, so the annual rate is 15,500,000, 11,500,000, 9,500,000 and 8,500,000 BZPX, and the cumulative emission at the end of each year is 15.5, 31, 42.5, 54, 63.5, 73, 81.5 and 90 million [README.md:320-327 @9958d5f]. These are scheduled amounts per deployment, not paid amounts: a window in which no position holds weight emits nothing, and no one inherits it.

YEAR	BASE (BZPX)	BONUS (BZPX)	SCHEDULED IN YEAR	CUMULATIVE AT YEAR END
1	7,500,000	8,000,000	15,500,000	15,500,000
2	7,500,000	8,000,000	15,500,000	31,000,000
3	7,500,000	4,000,000	11,500,000	42,500,000
4	7,500,000	4,000,000	11,500,000	54,000,000
5	7,500,000	2,000,000	9,500,000	63,500,000
6	7,500,000	2,000,000	9,500,000	73,000,000
7	7,500,000	1,000,000	8,500,000	81,500,000
8	7,500,000	1,000,000	8,500,000	90,000,000

TAB-25.3 — the table above, per deployment; the sums are exact in the idealised schedule, the integer closed form E27 differs by the wei amounts printed in the derivation.

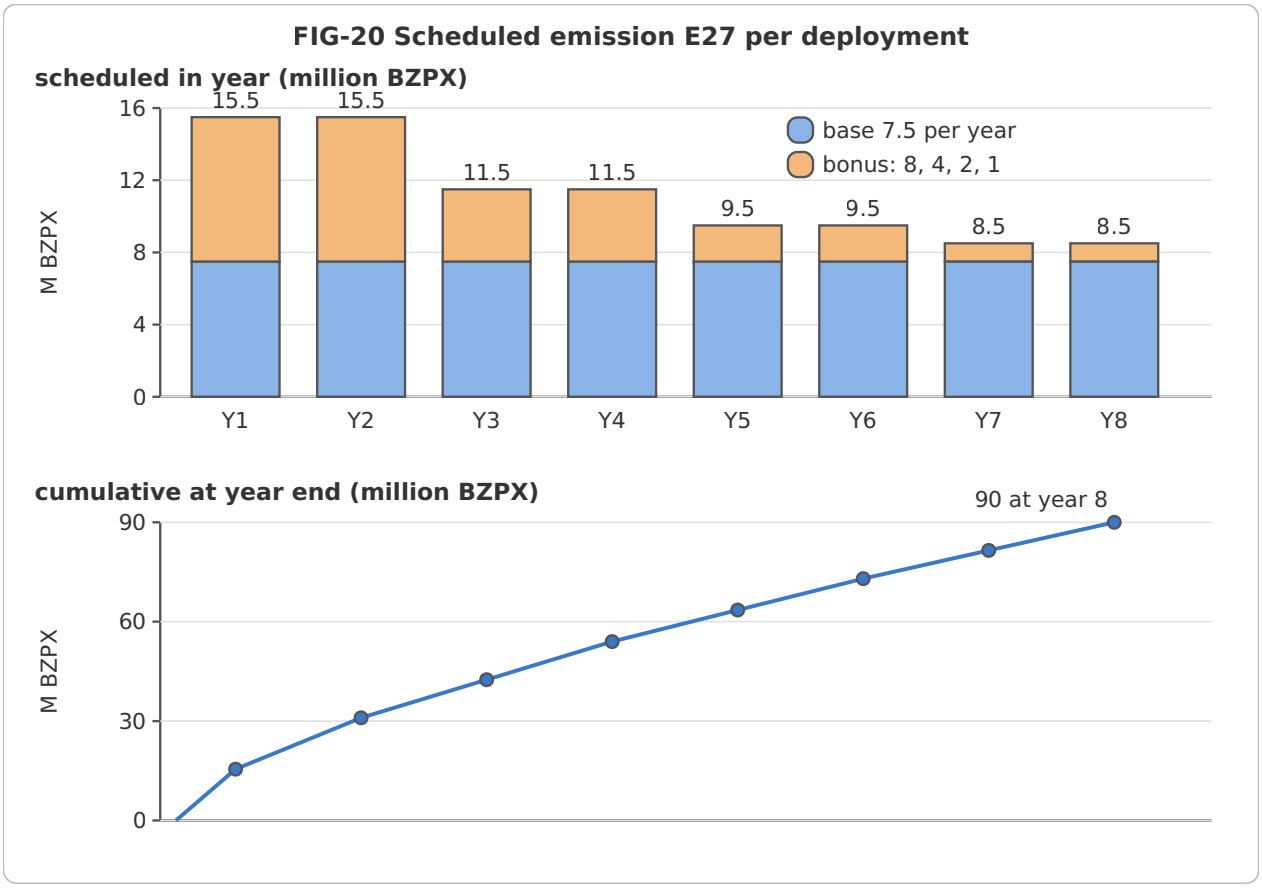

FIG. 20 Scheduled emission E27 per deployment

FIG-20 — Scheduled emission E27 per deployment against time: a straight base line of 7,500,000 BZPX per year under a bonus staircase that halves every two years (8M, 4M, 2M, 1M per year), the cumulative curve closing at exactly 90,000,000 BZPX after eight years; scheduled, not paid.

The window. At every entry point, with W = `totalBoostedEffective`, $W_{\min}$ = `MIN_EMISSION_WEIGHT` (1,000e18, L228), t_ℓ = `lastRewardTime` and $t' = \min(t, \texttt{emissionEnd})$, the elapsed window is paid into a per-weight accumulator A = `accRewardPerShare`. The window reward is computed by `_windowReward` (L1646–1650) and applied by `_updateGlobal` (L1620–1640):

$$r \;=\; \min\!\left(\left\lfloor \frac{(E(t') - E(t_\ell)) \cdot \min(W, W_{\min})}{W_{\min}} \right\rfloor,\ \texttt{rewardReserve}\right), \qquad A \;\leftarrow\; A + \left\lfloor \frac{r \cdot 10^{18}}{W} \right\rfloor \tag{E28}$$

`rewardReserve` falls by r and `totalRewardDistributed` rises by r. With $W = 0$ the clock moves and nothing is captured, so no one inherits a backlog; below $W_{\min}$ the window is throttled pro rata, so a dust position cannot take the schedule alone; the clock is frozen while the contract is in emergency; and because each window is capped at the reserve, `fundEmission` settles the clock before new funding lands, so a top-up never pays a window retroactively.

Settlement. A position with emission weight w_u and checkpoint c_u (`rewardDebt`) is settled in `_settleEmission` (L2141–2152): the gross entitlement is $g = \lfloor w_u A / 10^{18} \rfloor$, the amount earned is $g - c_u$, and any part of it that lies above the weight the position holds live (`_atLiveWeight`) was never earned by anyone: it returns to `rewardReserve` and leaves `totalRewardDistributed`. Then:

$$\mathrm{due}_u \;=\; \left(\left\lfloor \frac{w_u\,A}{10^{18}} \right\rfloor - c_u\right)^{+} - \mathrm{excess}_u, \qquad c_u \;\leftarrow\; \left\lfloor \frac{w_u\,A}{10^{18}} \right\rfloor \tag{E29}$$

`_settlePendingRewards` (L1699) adds any previously deferred amount and defers the payment, rather than reverting, if the contract's balance does not cover what is due. The pure-yield stream has the same shape with w_u^{p}, A^{p} and `pureYieldDebt`. After every change of weight the single writer re-checkpoints (`_resync` L2056, `_applyBoost` L2020, `_checkpoint` L2035), so a position earns only on windows that close after its weight is in place.

Boost. The weight is the position's stake times a boost that is a polynomial in the lock it is *currently* paid for. With ℓ the lock in days (zero once the unlock time has passed, whether or not the position has been normalised), capped at 2,555 (`MAX_LOCK_DAYS`, L242):

$$b(\ell) \;=\; 10^4 + \left\lfloor \frac{750\,\ell}{365} \right\rfloor + \left\lfloor \frac{250\,\ell^2}{365^2} \right\rfloor \text{ bps}, \qquad w_u = \left\lfloor \frac{(s_u - d_u)^{+}\,b}{10^4} \right\rfloor, \qquad w_u^{\mathrm{p}} = \begin{cases} \lfloor s_u\,b/10^4 \rfloor & d_u = 0 \\ 0 & d_u > 0 \end{cases} \tag{E30}$$

with s_u the position's stake and d_u its debt (constants `BOOST_BASE`, `BOOST_LINEAR`, `BOOST_QUAD`, L244–246). Exact values: $b(90) = 10{,}199$, $b(365) = 11{,}000$, $b(730) = 12{,}500$, $b(1{,}825) = 20{,}000$, $b(2{,}555) = 27{,}500$ bps, that is 1.10x at one year, 2.00x at five years and 2.75x at seven years [the 90- and 730-day values carried from 2.3, recomputed from the formula]. Emission weight is on net stake; the pure-yield weight exists only for a debt-free position: a borrower does not earn the interest it pays.

Worked example (illustrative, not a forecast). Two positions of 1,000 BZPX each, no debt, in a window where they are the only weight. Position X is locked 365 days (b = 11,000, weight 1,100); position Y is locked 1,825 days (b = 20,000, weight 2,000). Total weight 3,100. X takes 1,100/3,100 = 35.5 % of that window's scheduled emission and Y takes 64.5 %. The numbers are chosen for arithmetic only: the rate a real position earns depends on the total weight at the time, which is not known in advance, and this paper states no yield.

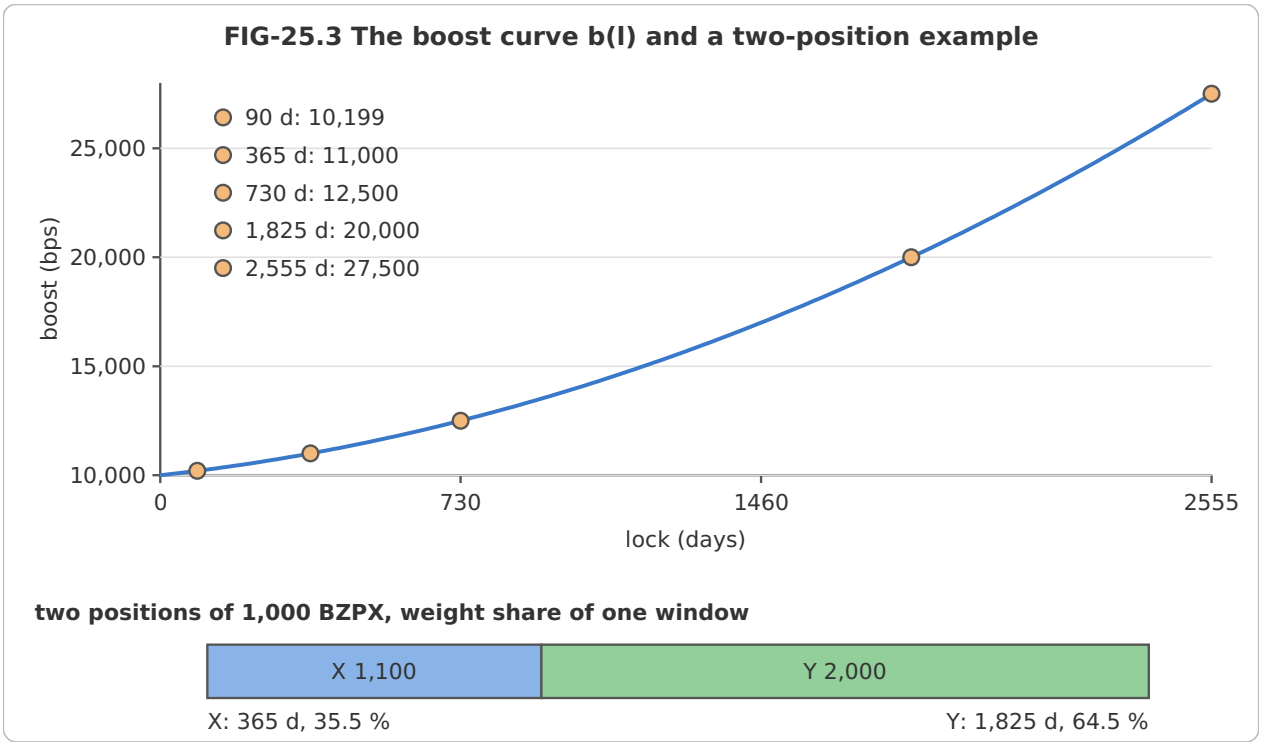

FIG. 25.3 The boost curve b(ℓ) and a two-position example

FIG-25.3 — the boost curve b(ℓ) from 90 to 2,555 days, with the marked points above, and the two-position example as a bar of weight shares.

§25.4 Interest: the curve and the index

The curve. Utilisation is $U = \min(\lfloor D \cdot 10^{18}/S \rfloor, 10^{18})$, with D = `totalDebt` and S = `totalStaked` ($U = 10^{18}$ if $S = 0 < D$; the base rate if both are zero), and the annual rate in bps is kinked at 80 % (`simulateRate` , L2623–2627):

$$r(U) \;=\; \begin{cases} 100 + \lfloor 500\,U/10^{18} \rfloor & U \le 0.8 \cdot 10^{18} \\ 500 + \lfloor 72{,}500\,(U - 0.8 \cdot 10^{18})/10^{18} \rfloor & U > 0.8 \cdot 10^{18} \end{cases}$$

(E31)

The two branches meet at the kink — $100 + 0.8 \times 500 = 500$ — and the curve reaches $500 + 0.2 \times 72{,}500 = 15{,}000$ bps at full utilisation. A borrow may not lift the book above 75 % (E41), so no borrow by itself enters the steep branch.

The index. Each elapsed slice Δt is stamped with the rate in force across it, before any write to D or S (`_interestSlice` , L1860–1890; `_updateInterestIndex` , L1818–1854), with Y = 365 days:

$$\delta \;=\; \left\lfloor \frac{\lfloor 10^{18}\, r(U)/10^{4} \rfloor \cdot \Delta t}{Y} \right\rfloor, \qquad \sigma \;=\; \left\lfloor \frac{D\,\delta}{10^{18}} \right\rfloor, \qquad I \leftarrow I + \delta, \quad S \leftarrow S - \sigma$$

(E32)

If σ would exceed S, the index advance is re-derived as $\delta = \lfloor S \cdot 10^{18}/D \rfloor$ and σ recomputed from it, so the per-position charges never sum to more than the global debit. The arithmetic saturates to zero instead of reverting, so the entry points that run beneath it cannot freeze, and under an emergency the clock slides without charging (L1826). One function computes the slice for execution and for every quote, so the two cannot round differently.

The split. The slice leaves the stake and is divided in the same step (L1836–1837; `_dampPure` , L1901–1903), with W_{p} = `totalBoostedPure` and A^{p} = `accPureYieldPerShare` :

$$\kappa_r = \left\lfloor \frac{300\,\sigma}{10^{4}} \right\rfloor, \quad \pi = \sigma - \kappa_r, \quad \eta = \left\lfloor \frac{\pi \cdot \min(W_{\mathrm{p}}, W_{\min})}{W_{\min}} \right\rfloor, \quad A^{\mathrm{p}} \leftarrow A^{\mathrm{p}} + \left\lfloor \frac{\eta \cdot 10^{18}}{W_{\mathrm{p}}} \right\rfloor, \quad \texttt{protocolReserve} \mathrel{+}= \kappa_r + (\pi -$$

(E33)

so $\sigma = \kappa_r + \eta + (\pi - \eta)$ exactly; with no pure weight, all of π goes to the reserve. The throttle is the same as emission's (E28), with the same constant.

Attribution. The global side was settled when the interest accrued; a position's own share is attributed when it is next touched (`_accrueInterestFor` , L1905–1954), with I_u its checkpoint of the index:

$$i_u \;=\; \left\lfloor \frac{d_u\,(I - I_u)}{10^{18}} \right\rfloor; \qquad i_u > s_u \;\Rightarrow\; \lambda = i_u - s_u \text{ enters E37},\; S \mathrel{+}= \lambda,\; i_u \leftarrow s_u; \qquad s_u \leftarrow s_u - i_u, \quad I_u \leftarrow I$$

(E34)

Interest is paid out of the stake; the debt itself does not grow. A debt-free position only re-checkpoints, so it never inherits index growth from a time it owed nothing. Interest that a position cannot pay was nevertheless distributed by E33; adding it back to S keeps $S = \sum_u s_u$, and the loss is booked by E37 at the moment it is known.

§25.5 The reserve, solvency and the loss waterfall

Conservation. With B the contract's BZPX balance, D = `totalDebt`, BD = `totalBadDebt`, S = `totalStaked`, RR = `rewardReserve`, PR = `protocolReserve` and Π = `totalRewardDistributed - totalRewardsPaid` (distributed, not yet paid), every function that wears the `conserves` modifier snapshots both sides before its body and reverts after it unless (L467–486; `_conservationSnapshot`, `_conservationCheck`, L497–505):

$$\left|\,\Delta(B+D+BD)\;-\;\Delta(S+RR+PR+\Pi)\,\right|\;\le\;\varepsilon,\qquad \varepsilon=\texttt{CONSERVATION_DUST}=10^{10}\text{ wei}$$

(E35)

Every term is a single non-negative state variable, summed without a floor, so the identity is linear in every regime — including terminal distress, where the recovery tools (`repay`, `liquidate`, `absorbBadDebt`) must keep working — and only the change across a call is compared, so history cannot accumulate into a false alarm.

Solvency. What the contract owes, and the public test of whether it holds it (`_owed`, L2096–2100; `_hardBreach`, L2184–2186; `isSolvent`, L2206):

$$\text{owed}\;=\;(S+RR+PR-D)^{+}\;+\;\Pi,\qquad \text{hardBreach}\iff B+\varepsilon<\text{owed},\qquad \texttt{isSolvent}()=\neg\,\text{hardBreach}$$

(E36)

Derivation of the link to bad debt. Where $S+RR+PR\ge D$, E35 gives $\Delta(B-\text{owed}+BD)=0$ (up to ε) across every guarded call, so $B-\text{owed}=c-BD$, with c the balance held beyond the ledger. Recorded bad debt is therefore exactly the term that pushes the book towards a hard breach, and the book reads insolvent once $BD>c+\varepsilon$. Under a hard breach the doors that would pay out at par or add new money to a short pot revert — `deposit`, `borrow`, `withdraw`, `claimRewards`, `claimPureYield` and the pure-yield settlement inside them, `withdrawReserve`, `cancelEmergency` (L898, L968, L1040, L1069, L1084, L1791, L595, L686) — while `repay`, `liquidate` and `absorbBadDebt` stay open, and anyone may call `tripBreaker` to move the contract into emergency (L660).

The waterfall. A loss λ arises at four sites only: a liquidation whose debt exceeds the stake (L1252–1257, L1279), an under-water emergency exit (L844–849), the write-off of an under-water position by `syncEquity` in an emergency (L748–751), and interest a position cannot pay (L1938–1946). All four apply the same order:

$$\kappa\;=\;\min(\lambda,\;PR),\qquad PR\leftarrow PR-\kappa,\qquad BD\leftarrow BD+(\lambda-\kappa)$$

(E37)

The reserve — built from the reserve factor of every interest slice (E33) and from the liquidation bonuses it withholds (a self-liquidation, or any liquidation under a hard breach) — absorbs first; only the uncovered rest becomes recorded bad debt. Recapitalisation is a public door that takes the reserve first, then at most what the caller offers and never more than is still uncovered (`absorbBadDebt`, L711–720), with a the caller's amount:

$$\rho\;=\;\min(BD,\;PR),\qquad \gamma\;=\;\min(a,\;BD-\rho),\qquad PR\leftarrow PR-\rho,\qquad BD\leftarrow BD-\rho-\gamma,\qquad B\leftarrow B+\gamma$$

(E38)

It reverts when $\rho+\gamma=0$, moves nothing out and pays the caller nothing; E35 holds by construction ($\Delta\text{LHS}=\gamma-\rho-\gamma=-\rho=\Delta\text{RHS}$). The loss of an under-water position reaches BD once it is recorded: by its own exit or liquidation or, in an emergency, by a public `syncEquity` call that writes it off through E37

(L733–761). Once BD is back within the dust, `isSolvent()` reads true and the admin may cancel the emergency.

Liquidation. A position is liquidatable iff $d_u > 0$ and ($s_u = 0$ or $100\,d_u \geq 95\,s_u$) (`_isLiquidatable`, L1284–1288). Closing it (`_executeLiquidation`, L1225–1282):

$$\text{seized} \;=\; \min\!\left(d_u + \left\lfloor \frac{500\,d_u}{10^4} \right\rfloor,\; s_u\right), \qquad \text{bonus} \;=\; (\text{seized} - d_u)^+, \qquad \lambda \;=\; (d_u - s_u)^+ \text{ to E37} \tag{E39}$$

The bonus goes to the caller, except when a borrower closes its own position or the book is in hard breach, in which case it stays in `protocolReserve`. The position's earned streams are paid before it is closed, and any stake left over remains as a debt-free position that keeps its lock.

Emergency exit. With pot $= B$ and E_+ = `totalPositiveEquity` $= \sum_u \max(s_u - d_u, 0)$ as tracked, the exit pays (`emergencyWithdraw`, L772–869; the scaling, L827–835):

$$x_0 \;=\; (s_u - d_u)^+, \qquad E' \;=\; E_+ - e_u + x_0, \qquad x \;=\; \min\!\left(\begin{cases} \lfloor x_0\,\text{pot}/E' \rfloor & \text{pot} < E' \\ x_0 & \text{otherwise,} \end{cases},\; \text{pot}\right) \tag{E40}$$

where e_u is the exiter's tracked term, replaced by its exact equity after accrual. On a solvent book the scaling never engages and net equity is paid in full; on a short one each exit takes the same fraction of its claim, and an under-water exit books its shortfall through E37 (L844–849).

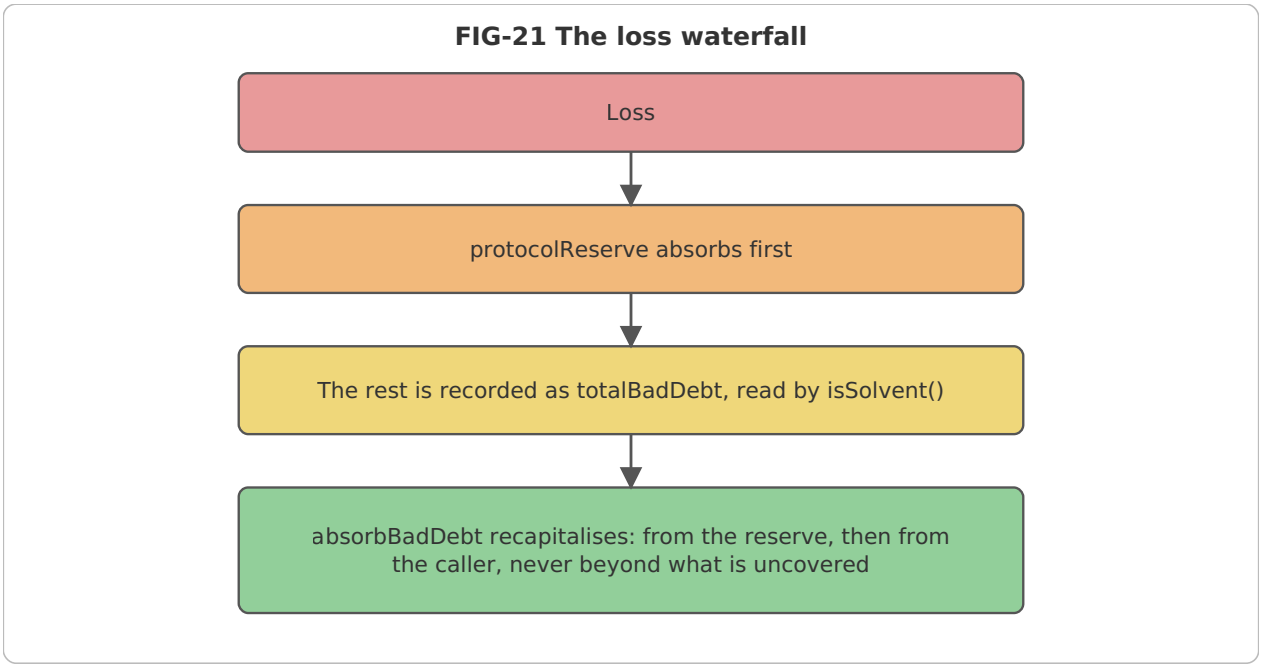

FIG. 21 The loss waterfall

FIG-21 — The loss waterfall E37–E38: protocolReserve absorbs first, the rest is recorded as totalBadDebt and read by isSolvent(), and absorbBadDebt recapitalises from the reserve, then from the caller, never beyond what is uncovered; illustrative proportions, no token amounts.

Borrowing ceilings. A borrow of a must satisfy both (`borrow`, L974–980; `_ltvCap`, L2305–2306):

$$d_u + a \;\leq\; \left\lfloor \frac{50\,(s_u - d_u)^+}{100} \right\rfloor, \qquad \left\lfloor \frac{(D+a)\cdot 10^{18}}{S} \right\rfloor \;\leq\; 0.75 \cdot 10^{18}$$

(E41)

the first on the position's net stake, the second on the whole book, held below the 80 % kink of E31.

Maintenance. Each user transaction, unless the contract is paused or in emergency, also scans a window of the borrower registry (liquidating what is under water) and of the lock registry (normalising expired locks), each of size (`_windowBudget` , L1299–1308; `_autoMaintain` , L1332–1360):

$$c(n) \;=\; \min\!\left(1 + \left\lfloor \frac{n}{50} \right\rfloor + \left\lfloor \frac{t - t_m}{15\ \text{min}} \right\rfloor,\ 10,\ n\right)$$

(E42)

with n the registry size and t_m the last sweep; at most four lock normalisations run per transaction (L283). The schedule is a pure function of on-chain state — no keeper, no governance — and its gas is bounded in every case.

Invariants. The design adds nine invariants, INV-23 to INV-31 (Appendix B): the conservation identity, the emission ceiling, the loss order, the closed doors under a breach, the recapitalisation bound, the single writer of the weights, the borrowing ceilings, the bounded emergency exit, and the deposit ageing rule.

§25.6 Hardening in V4

Four merged changes shaped the V4 source after the design that 2.3 described; each is described here at the level of the design it produced.

- **#22 (`b44e238` , 2026-10-11).** Interest is stamped per slice by one function shared by execution and every quote (E32), so a preview and a transaction compute the same charge, at the rate in force across the slice. The pure-yield channel is throttled by the same weight floor as emission (E33), so a residual denominator cannot capture the flow. A borrower closing its own position receives no liquidation bonus. Read-only risk views were added (`healthFactor` , `daysToLiquidation` , `maxBorrowOf`).
- **#26 (`7fc0c2a` , 2026-10-13).** One loss order for the three loss sites of the time (liquidation, under-water emergency exit, unpaid interest; the fourth, the `syncEquity` write-off of §25.5, came with #42): the reserve first, then recorded bad debt (E37); a public recapitalisation door (E38); the value-moving doors closed under a hard breach (E36); a constructor that refuses a token address with no code; the conservation check emitted once as two internal helpers instead of being inlined at every guarded site, with the identity unchanged; and an assurance test suite.
- **#25 (`cc614f9` , 2026-10-14).** Documentation only: short summary lines in the guides and test headers. No contract source changed.
- **#42 (`02af63a` , 2026-11-16).** In an emergency, `syncEquity` writes off a position whose stake no longer exceeds its debt, as that position's own exit would: the shortfall goes through the loss order of E37, the reserve first and the rest to recorded bad debt, where `absorbBadDebt` can reach it, so a recapitalised book can read solvent and the emergency can be cancelled. A position above water, or without debt, is only re-checkpointed.

§25.7 Test evidence

The staking repository declares 22 stateful invariants in 6 files at `9958d5f` (checked by listing every `invariant_*` function of the test tree): 7 in `BlazePhoenixStaking.t.sol` , 9 in `InvariantsFullSurface.t.sol` , 2 in `assurance/EmissionScheduleMatrix.t.sol` , 1 in `atq/AtqEmergencySurface.invariant.t.sol` , 2 in

`attack/EmissionShareInvariant.t.sol` and 1 in `hunt/ExactAttributionInvariant.t.sol`. Solvency is a named invariant, not a comment: `invariant_solvent` (the contract's own `isSolvent`) and `invariant_backingCoversOwed` (the real BZPX balance covers every ledger term the contract owes), the second stated twice, in the main file and in the full-surface file as `invariant_INV16_backingCoversOwed`. The emission side has its own two: the distributed total never exceeds the schedule and the schedule never exceeds `TOTAL_REWARDS` (`invariant_DistributedNeverExceedsTheScheduleWhichNeverExceedsTotalRewards`), and the reserve plus the distributed total is always what was funded (`invariant_ReservePlusDistributedIsAlwaysWhatWasFunded`). The full list by name is Appendix B. These are counts of what is declared; this paper does not report a pass rate or a mutant count for the staking suite. The number of `test*` functions, of test files and of mutants at `9958d5f` is not reported here; the 2.3 figures (37 files, 256 `test*`, 14 `invariant*` at `cc614f9`) are not carried.

§25.8 Two deployments, one schedule each

The staking contract is deployed twice, and the two deployments are independent. The repository describes them as "two independent deployments of this contract, each capped at 90,000,000 BZPX" (README.md:290, 306–310 @9958d5f): each has its own contract, its own reward reserve, its own protocol reserve and its own emission clock. Nothing crosses from one to the other: a position, a debt, a reserve or a breaker state in one deployment is invisible to the other, and a trip, an emergency or a haircut in one does not touch the other. The deploy script funds one contract per run: it requires the deployer to hold at least 90,000,000 BZPX and funds the emission atomically with the deployment (`script/Deploy.s.sol:5-12, 54, 79, 92-98` @9958d5f), and it does not fix the chain, so the same bytecode can be placed on two networks.

The sum of the two caps, 180,000,000 BZPX, is the staking line of the token allocation in §20. It is a sum of two separate budgets, not a single pool: the contract does not know about its twin. A reader who wants the schedule of one deployment reads TAB-25.3; the schedule of the other is the same table, on its own clock.

PROPERTY	DEPLOYMENT 1	DEPLOYMENT 2
Emission cap (`TOTAL_REWARDS`)	90,000,000 BZPX	90,000,000 BZPX
Emission length	8 years from its own `emissionStart`	8 years from its own `emissionStart`
Reward reserve, protocol reserve	its own	its own
Circuit breaker and emergency state	its own	its own
Shared state	none	none

The two deployments hold 90,000,000 BZPX each; their networks are named at launch, not in this paper. The sum, 180,000,000 BZPX, is the staking line of §20. The staking engine is pre-launch: the site gives its launch as Q2 2027, a schedule and not a deployment, and this paper gives no staking address because there is none yet.

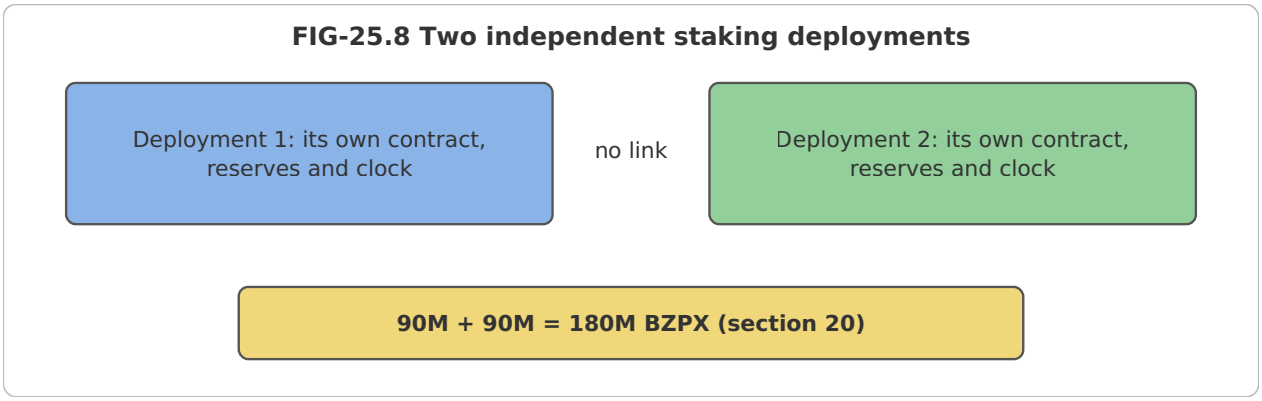

FIG. 25.8 Two independent staking deployments

FIG-25.8 — two side-by-side boxes, each with its own contract, reserves and clock, no arrow between them; underneath, the 90M + 90M = 180M line of §20.

§25.9 Clocks: seconds, blocks and chains

The contract keeps time in two units, and a reader should know which quantity uses which. **Seconds** (`block.timestamp`): the emission schedule (E27), the interest index (E31), the lock and its expiry, the emergency timestamp (`emergencyTrippedAt`) and the maintenance gap (`MAINT_GAP_UNIT` = 15 minutes, L274). **Blocks** (`block.number`): exactly one guard, the same-transaction and flash-loan guard `MIN_DEPOSIT_BLOCKS` = 10 (L248): a deposit must age more than ten blocks before `withdraw`, `claimPureYield`, `lock` or a debt-free reset can act on it (for example L1031, L1083, L1112). Consequently the *wall-clock* length of that guard depends on the chain's block time, while every economic quantity runs on seconds and is the same on any chain. The wall-clock time of ten blocks differs by chain and is not asserted here.

QUANTITY	CONSTANT	UNIT	VALUE
Emission period	`HALVING_PERIOD` (L210)	seconds	2 × 365 days = 63,072,000 s
Emission length	`EMISSION_LENGTH` (L212)	seconds	4 periods = 252,288,000 s (8 years)
Lock bounds	`MIN_LOCK_DAYS`, `MAX_LOCK_DAYS` (L241–242)	days	90 and 2,555
Deposit age guard	`MIN_DEPOSIT_BLOCKS` (L248)	blocks	10
Emergency grace	`EMERGENCY_GRACE_PERIOD` (L269)	seconds	30 days (retained for telemetry and UX, per its own comment)
Maintenance gap	`MAINT_GAP_UNIT` (L274)	seconds	15 minutes

The period is 2 × 365 days, not two calendar years: leap days are not counted, so "eight years" in this chapter means 8 × 365 days = 2,920 days. The emergency clock behaves as in §25.2 and §25.11: the interest index and the emission clock freeze when emergency mode is entered, and `cancelEmergency` slides both clocks to the

current time, so the halt is never charged or emitted retroactively. The reasoning behind treating time as a first-class quantity of an invariant is in `docs/INVARIANTS_AND_TIME.md` (an essay in the repository, not a catalogue; §10 applies it to this emission).

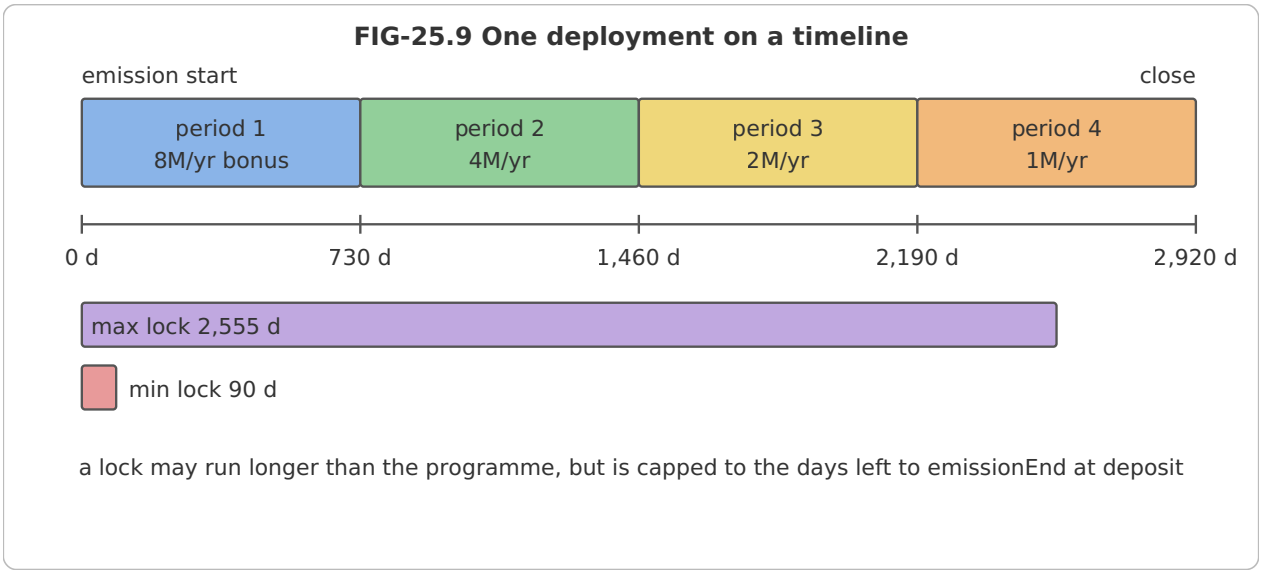

FIG. 25.9 One deployment on a timeline

FIG-25.9 — a timeline of one deployment: emission start, the four two-year periods and the close at 2,920 days, with the lock bounds (90 d and 2,555 d) drawn as bars against it, to show that even the longest lock (2,555 d) is shorter than the programme, and that at deposit a lock is further capped to the whole days left to `emissionEnd` *(* `maxLockDaysAvailable()` *; a longer request reverts), so no lock outlives the programme.*

§25.10 Reading solvency yourself

Solvency is a public view, free to call, and it is built so that a reader does not have to trust a dashboard. Four functions answer, in increasing detail.

`owed()` returns every claim on the contract, recorded bad debt included and never netted; `backing()` returns the physical BZPX the contract holds. `isSolvent()` is true exactly when `backing() + dust ⩾ owed()`: it is `!_hardBreach()` (L2206), with `CONSERVATION_DUST` = 1e10 wei (L267). The relationship the other direction is the trip condition: `isSolvent()` is false exactly when `tripBreaker` would succeed outside an emergency. A collateralisation ratio in WAD (`backing × 1e18 / owed`, floored; 1e18 means exactly covered) is published by a single producer, and `solvency()` reports that reading rather than a copy of it. `owed()` is a lower bound until the next sweep: a loss not yet recorded (a borrower whose accrued debt has passed her stake) is read only in part until a sweep, a liquidation, the borrower's own emergency exit or, in an emergency, a public `syncEquity` call records it, which is documented in `docs/KNOWN_ISSUES.md`. A view that reads true therefore means "solvent on what has been recorded", and the paper says so.

`auditInvariants()` (L2270) returns a bitmask, one bit per check, 0 meaning all hold: bit 0 hard breach (balance + dust < owed); bit 1 a soft solvency signal (debt above 95 % of stake, or stake zero with debt outstanding); bit 2 `rewardReserve` above `TOTAL_REWARDS`; bit 3 a monotonicity snapshot of the two accumulators

(raised by the permissionless `touchAuditSnapshot`, which moves no value); bit 4 boost bounded: the weights must be covered by the stake they were built on. Only bit 0 gates `tripBreaker`; bit 1 is a signal, not a guard, because a position may be transiently unhealthy before a liquidation.

The modifier `conserves` (L467) is the mechanism behind all of it. Around every value-moving transaction it compares the change of the two sides of the master identity, each side a plain sum of non-negative state variables with no $\max(x, 0)$ clamp, so it stays linear in every regime, `balance + totalDebt + totalBadDebt + stakedOverdraw == totalStaked + rewardReserve + protocolReserve + pendingDistribution`, within dust, and reverts if a transaction leaks value. `emergencyWithdraw` and `syncEquity` sit outside it so they stay callable in a breach. The identity is checked on the *change* across the call, so historical rounding drift cancels and only a transaction that itself leaks value fails.

Try it (a procedure, not a result). With any RPC of your choice, call `owed()`, `backing()`, `isSolvent()` and `auditInvariants()` on the staking contract at a block of your choice, and compare `backing() + 1e10` with `owed()` yourself. The four values and the block number are all you need; nothing here depends on our servers. `solvency()` (L2242) returns a `SolvencyReport` struct.

BIT	NAME (IN THE SOURCE)	MEANING	GATES `TRIPBREAKER`?
0	Conservation	balance + dust < owed (hard breach)	yes
1	Solvency	debt > 95 % of stake, or stake 0 with debt	no, a signal
2	EmissionCap	`rewardReserve` > `TOTAL_REWARDS`	no
3	Monotonicity	an accumulator fell below its snapshot	no
4	BoostBounded	weights exceed the stake they were built on	no

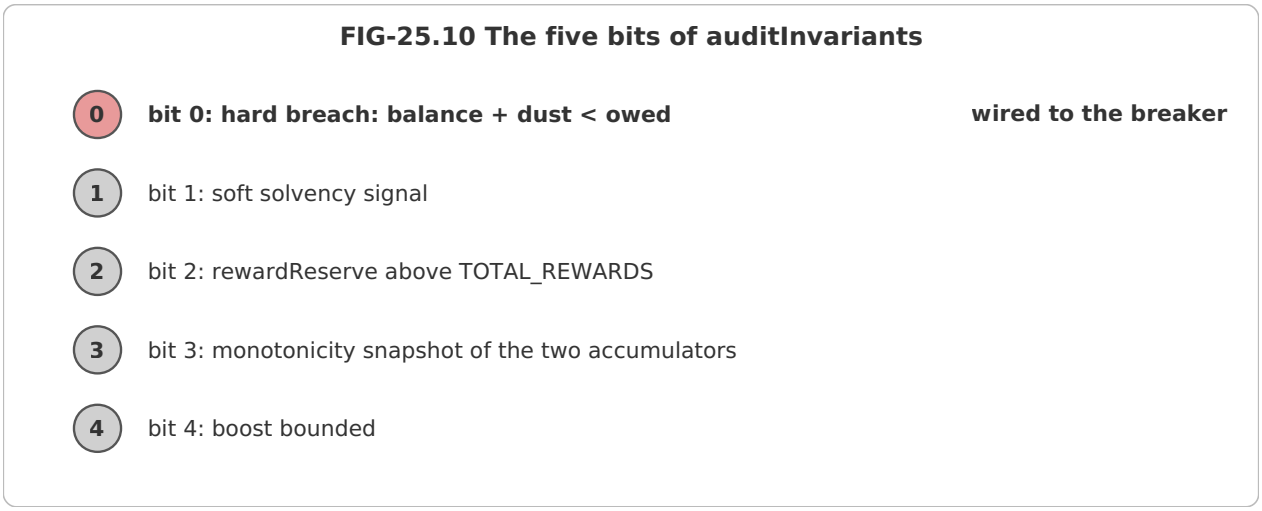

FIG. 25.10 The five bits of auditInvariants

FIG-25.10 — the five bits of `auditInvariants` as five lamps, bit 0 highlighted as the only one wired to the breaker.

§25.11 The circuit breaker

The staking contract has an emergency mode that stops new risk and lets stakers leave. There are two ways to enter it, and one way out.

Circuit breaker. Anyone can call `tripBreaker()`, but only when the contract's BZPX balance plus dust is below what it owes. The guardian can call `declareEmergency()` at its discretion. Neither call moves funds: both pause deposit, borrow, lock and liquidate, and open `emergencyWithdraw`, which applies a haircut. Only the admin can call `cancelEmergency()`, and only once the invariant holds again.

Exact definition, from the source at `9958d5f` *(read line by line).* `tripBreaker()` (L660) can be called by anyone, but reverts with `Staking__NoBreach` unless `_hardBreach()` is true, that is, unless the contract's real BZPX balance plus `CONSERVATION_DUST` (1e10 wei) is below `_owed()` (L2184); it reverts if the contract is already in emergency. `declareEmergency()` (L668) belongs to the guardian role and is discretionary: it does not depend on the breach condition. Neither moves funds. Both run the same body (`_enterEmergency`): the contract brings interest and emission up to the halt, then freezes both; it sets the emergency flag and the trip time; it records `totalRewardDistributed` as the anchor; it pauses the contract if it is not paused; and it emits `EmergencyDeclared` with a flag that says whether the trip was the permissionless breaker. While paused in emergency, `deposit`, `borrow`, `lock` and `liquidate` are closed; `repay` is also closed in emergency, and `emergencyWithdraw` is the only exit (see the table in §25.2). `emergencyWithdraw()` (L772) pays the position's net equity (stake less debt) and forfeits its unclaimed streams. The haircut applies only in a breached emergency: if the contract's balance is below the aggregate positive equity, each exit is scaled pro rata by balance ÷ aggregate equity, and the scaling is arranged so that the sum of payouts never exceeds the balance in any order of exits; if the contract is in emergency but holds enough, the full net equity is paid and the haircut does not engage. `cancelEmergency()` (L684) belongs to the admin role and reverts if the contract is not in emergency or if `_hardBreach()` is still true; it also refuses if the distributed total moved during the halt (a meta-guard), and on success it slides both clocks to now and clears the flag. The contract stays paused after `cancelEmergency`; the admin calls `unpause()` separately (L642), which itself is not available during an emergency.

The reason the trigger can be permissionless is stated in the source: nobody can lower the contract's balance except through flows that lower `_owed()` by the same amount, and donations only raise the balance, so the condition can fire only on a genuine shortfall, never on a healthy book, and it is reversible by the admin once the reading clears. The worst a spurious trip can do is a temporary pause that the admin can undo. This is the property that `invariant_INV20_breakerCannotTripWhileSound` (`InvariantsFullSurface.t.sol:421` @9958d5f) tests, and, once the contract is deployed (coming soon), the reader can check the same condition on chain with `isSolvent()` (§25.10).

CALL	WHO	CONDITION	MOVES FUNDS	EFFECT
`tripBreaker()`	anyone	balance + dust < owed (`_hardBreach`)	no	enters emergency
`declareEmergency()`	guardian	at its discretion	no	enters emergency
`emergencyWithdraw()`	a position holder	emergency active	its own exit only	pays net equity, pro rata if the balance is short

CALL	WHO	CONDITION	MOVES FUNDS	EFFECT
`cancelEmergency()`	admin	emergency active, `_hardBreach()` false, and no emission distributed since the trip	no	leaves emergency; contract stays paused
`unpause()`	admin	not in emergency	no	reopens the doors

Illustration of the haircut (arbitrary numbers, not a forecast). Suppose a breached emergency in which the contract holds 80 BZPX and the positions' aggregate net equity is 100 BZPX. Every exit is scaled by 80 ÷ 100 = 0.8, so a position of net equity 10 receives 8, whichever order the positions leave in. If the contract held 100 or more, the position would receive 10.

This is a statement of what the code does when a shortfall exists. It is not a statement that a shortfall cannot exist, and it is not insurance: the haircut is the contract's rule for dividing a loss, not a promise to avoid one. No external audit has been done, and the repository's pre-launch notice applies (§25 intro).

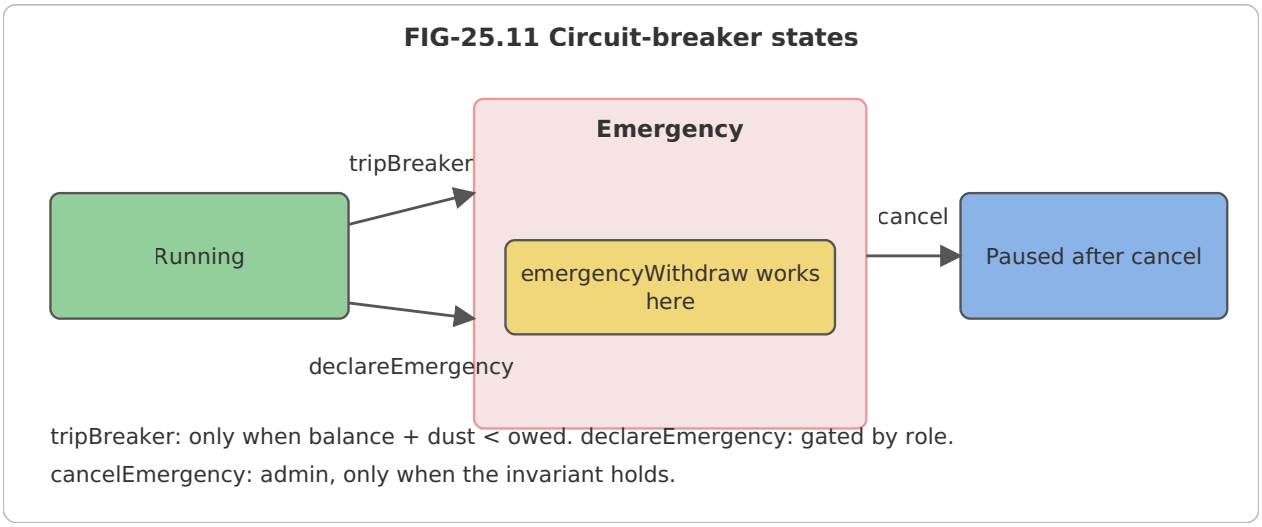

FIG. 25.11 Circuit-breaker states

FIG-25.11 — a state diagram with three states (running, emergency, paused after cancel), the two entry doors (`tripBreaker` gated by balance + dust < owed; `declareEmergency` gated by role) and the one exit door (`cancelEmergency`, admin, only when the invariant holds), with `emergencyWithdraw` drawn inside the emergency state.

Part VII — The token, the research programme and what you must 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 deployments of 90,000,000 each (§25.8); the V4 emission design is in §25
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

The allocation above is the one the project serves at `/facts.json`.

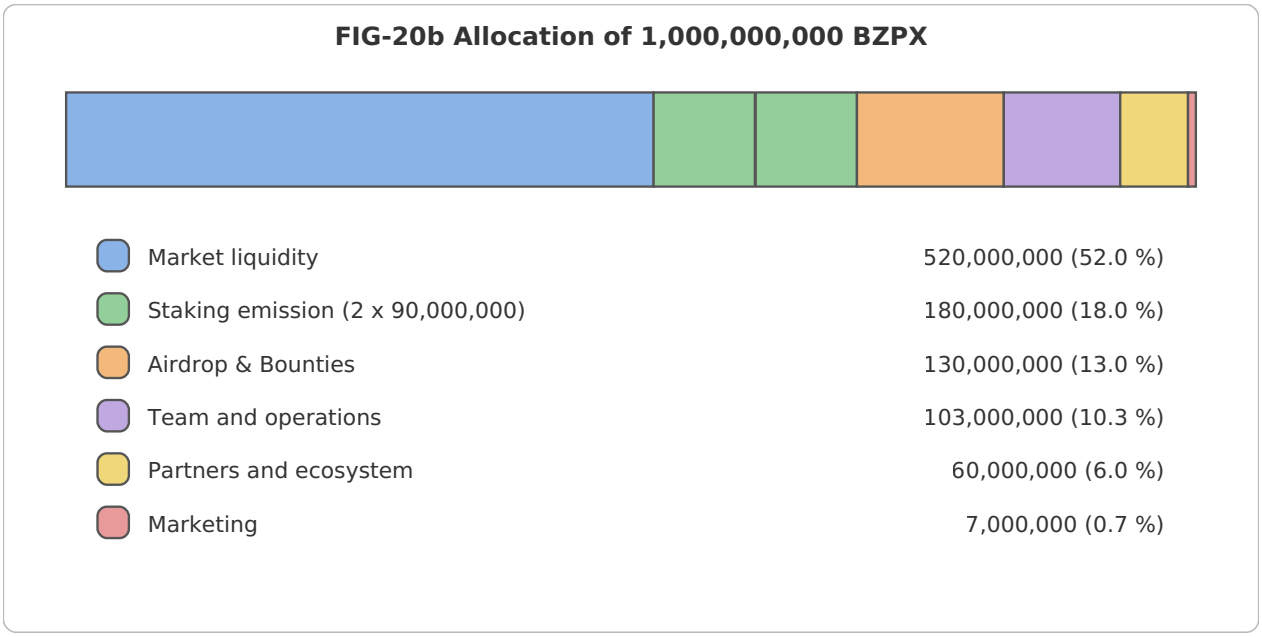

FIG. 20b Allocation of 1,000,000,000 BZPX

FIG-20b — the allocation as a single stacked bar of 1,000,000,000 BZPX with the six segments of TAB-20; the staking segment is two halves of 90,000,000.

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. (Historical entry, dated; kept as in 2.3.)

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

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. These are the names credited in 2.3.1; the repository's `SECURITY_HALL_OF_FAME.md` at `6387350` now credits seventy-six researchers, and is the list of record. No report has reached the Critical class. Attribution is never rounded down: names are published exactly as given.

Security awards are paid at the bands below, which are unchanged from 2.3.

SEVERITY	AWARD
Critical — direct theft or permanent freezing of user funds	2,500,000 – 7,500,000 BZPX
High — theft under specific conditions, or protocol insolvency	625,000 – 2,500,000 BZPX
Medium — griefing, temporary denial of service, value leakage	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 the 15 December 2026 launch and reports triaged now. One clause is deliberate: a report need not break a conservation law to be in scope — 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 asks to be trusted somewhere; almost none can enumerate where. 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 transfer itself (`setAdmin`), repoint both treasury slots (`setTreasuries` , which also redirects any fee the slots are owed), 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 (a rescue never takes the fees the slots are owed); 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`). The only balance the Router holds at rest is fee owed to a treasury slot: a fee push the token refuses is booked in `owed[token][slot]` instead of reverting the swap.

- **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** — two fee slots, set at construction (a zero argument falls back to the admin). Before renunciation the Router's control key can repoint both with `setTreasuries`, which refuses the zero address; `renounceControl` (refused while paused) freezes that for ever. Each treasury can always move its own slot, and only its own, with `moveTreasury`, also after renunciation, so a payee the fee token blacklists is not stuck. A fee the token refuses to push is booked to the slot in `owed[token][slot]` and pulled by the slot's current holder with `claimFee`. The 30 / 70 split is a compile-time constant with 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.
- **The Gate: external venues** — Balancer, Curve and the other venues that are reached through an adapter are reached only through an immutable wrapper in front of a curated adapter whose codehash is pinned; adding or replacing one waits `ADAPTER_DELAY` (48 hours) and a pair has at most four wrappers; a hostile venue's worst case is still bounded by your minimum and the floor (§14.4).
- **The site and the app** — add no trust to the trade: the quote is reproducible on your own RPC and the swap is bounded by your minimum, whatever the page displays (§14.5, §27).

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 settles no swap without paying the fee or booking it to the treasury's slot; 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.

> ***License: source-available, not open source.*** *The DEX source is published under the Business Source License 1.1 (BUSL-1.1). You may read it, build it and run it to check what we claim, but BUSL-1.1 is not an open-source license and this paper never calls it one. Licensor: BlazePhoenix Protocol. Change License: GNU General Public License v2.0 or later. Change Date: 2030-06-01 (README.md:47 and LICENSE:5, 11, 12 at* `6387350`*). The license text converts on the Change Date or on the fourth anniversary of the first publicly available distribution of the code, whichever comes first (LICENSE:37–40). Production use before conversion is limited by the license's Additional Use Grant, which you should read in the repository rather than in this paper.*

§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 0.01 bps of an exhaustive 10 % split grid on Arbitrum (−0.0072 bps), Optimism and Robinhood Chain (equal to the wei), and 1.178 bps above it on Base, where the 10 % grid is therefore not the optimum (§17.2, `969bcc6`). 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, which this protocol does not price in its Core and reaches only through the Gate, behind a code-pinned adapter (§14.4); 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 intro-

duced mutation analysis, whose coupling hypothesis the curated guard of §15.2 relies on. Avizienis's N-version 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 1,859 and 2,796 bytes under (1,283 and 2,220 under the project's own 24,000-byte guard); 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

This paper has described one machine and one discipline, and the discipline is the part that travels. Nothing that a stranger can supply is allowed to choose its own shape, its own domain or its own floor (§2); a number that comes from the caller is measured, re-homed, or made costly to lie about; every guard has a mutant that turns it red (§15.3); and the question "what is this quantity?" has exactly one producer (§2.5).

The same habit applies to the text. Every number here carries its denominator and the command that reproduces it (§15.1), and what none of the evidence establishes is said in plain words (§15.11). Where this paper could not measure something, it says so in place, rather than rounding the gap into a claim. A reader who finds a number without a command, or a sentence that outruns its evidence, has found a defect in the paper, and we want the report.

The software is not finished by being described. The DEX source is source-available under BUSL-1.1, not open source; the staking engine is a pre-launch engineering preview. No external audit yet. Inspect it yourself: the code, the tests, the mutants, the bounty.

Revision history

Version 2.4.0 (2026-11)

2.4.0 describes the V2 generation of the DEX at `6387350` and the Staking V4 engine at `9958d5f` (measurements stay dated at `969bcc6`, 2026-11-16); the V1 generation is described only as the deployed one, in §19. What changed against 2.3.0, each item checked against the commit named:

1. Route shape: at most 2 hops (`MAX_HOPS = 2` , was 3) and at most 5 legs per hop (`MAX_LEGS_PER_HOP = 5`), so a route has at most 10 legs (2 hops × 5 legs, was 11); the solver has 4 topologies, the direct route and three through a bridge (`TOPOLOGIES = 1 + BRIDGE_SLOTS = 4` , was 13).
2. Architecture: the Router, Solver, Hub, Quoter and Core are joined by the Gate, which puts every external venue behind an immutable per-venue wrapper (Balancer V2 and V3, Curve StableSwap-NG, Twocrypto and Tricrypto-NG, among others), by the Math library and by the adapters. The 2.3 state-

ment that Curve and Balancer were excised is withdrawn (§18). The deployed-size table (§16) lists 16 artifacts, with a total of 167,321 bytes of runtime.

3. Assurance: 54 stateful invariants in 18 files, 4 Certora specifications and 1,895 mutants (Appendix B), as stated in the repository README at `969bcc6`, which the invariant listing of Appendix B matches; 15 Halmos checks report `[PASS]` in the CI run at commit `6387350` (Appendix B.3).
4. Staking V4: the emission is 90,000,000 BZPX over eight years per deployment, a 60,000,000 BZPX linear base plus a 30,000,000 BZPX series that halves every two years (§25.3, E27); the equations E27–E30 are restated and the line numbers of E31–E42 are those of `9958d5f`; INV-24 is restated: the schedule closes exactly, at `BASE_REWARDS + S - (S >> EMISSION_PERIODS) = TOTAL_REWARDS` with remainder 0 wei.
5. Measurements: §17 keeps the 2.3 measurements with their date and commit, and adds the lines measured on 2026-11-16 at `969bcc6`.
6. Interfaces: a new section, §27, describes six ways to reach the DEX, each read at a named commit: the SDK (1.6.0 on npm at `bb41243`; main `2d2fc39`), the local MCP server (1.3.0, `3514804`), the hosted MCP and the API (`01c5468`), the Hermes plugin (1.0.0, `cf38638`) and the APP, with who performs the RPC and what each verifies (TAB-27.1, FIG-27).
7. Changes merged after `969bcc6`, each with its regression test: the fee-on-transfer leg ratio is measured, not recovered from a clamped slot (#137); the impact ceiling asks the leg's own book before refusing (#139); the funnel reads each book under the bar before dropping it (#138); the stable Newton exit steps down while the wei below still holds K (#141); the route floors read the declared gross, `outCap` (#132); the encode doors ask the Router's route protocol floor (#130); the release Yul step sequence drops the FunctionSpecializer, which changes the deployed sizes (#136). In the staking engine, after `c0366cf`: `syncEquity` writes off a position whose stake no longer exceeds its debt (#42).
8. Status: the V2 DEX and the staking engine are pre-launch engineering previews. No external audit yet.

Version 2.3.0 (2026-10-14)

2.3.0 — adds the Staking V4 design chapter (§25), with equations E27–E42, invariants INV-23–INV-31 and figures FIG-19–FIG-21; it supersedes the statement below that the staking engine is no longer described in this paper. Elsewhere only the version labels, the enumeration ranges, the appendix introductions and the DOI lines change; the rest is the 2.3 text.

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 2.2 edition, E23–E26 were added in 2.3, E43–E52 in 2.4 (their DEX line numbers at `6387350`), and E27–E42 are the staking engine's (§25): "Staking" in the fourth column names `src/BlazePhoenixStaking.sol` of the staking repository at `9958d5f`, with its line numbers at that commit. E27–E30 are restated for the 2.4 schedule of 90,000,000 BZPX over eight years per deployment.

#	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	$k(x, y) = x^3 y + x y^3$	Core `_solK`	§6

#	NAME	EQUATION	DEFINED IN	§
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 reach or cross zero; exit at $\lvert y_{n+1} - y_n \rvert \le 1$, then up to two one-wei climbs while $k < K$ (a third returns the seed) and up to two one-wei descents while the wei below still holds K; cap 64 → seed → quote 0	Core `_solY`	§6
E7	Decimal normalisation	$X = r_{\text{in}} 10^{18-d_{\text{in}}}$, $Y = r_{\text{out}} 10^{18-d_{\text{out}}}$, de-scaled after the solve	Core `_solidlyStable`	§6
E8	CREATE2 derivation	$\text{pool} = \text{addr}_{20}(\text{keccak256}(\texttt{0xff} \Vert \text{origin} \Vert \text{salt} \Vert \text{initCodeHash}))$	Core `create2Address`, `deriveAddress`	§7
E9	Monoslot packing	$s = a \mid f \ll 8 \mid k \ll 32 \mid r \ll 40 \mid (c \wedge \texttt{0xFFF}) \ll 48 \mid b \ll 60 \mid \ldots \mid \tau \ll 224$, with τ the pool's own clock (EMA in seconds between anchored ticks, 0 = not yet measured)	Core `encodeSlot`	§8
E10	Vitality Field	$\Psi = V \cdot 2^{b} \cdot (1 + 0.25\,\mathbf{1}_{\text{bridge}}) \cdot (1 + 0.05\,\mathbf{1}_{\text{conc}})$	Core `psi`	§9.2
E11	Activity decay	$V = \max(1, \lfloor c \cdot 24{,}576/h \rfloor)$, $c = (\text{swapCount} \gg \sigma) - \lfloor c\,(a \bmod h)/(2h) \rfloor$, $\sigma = \lfloor a/h \rfloor$, $a = t - t_{\text{last}}$, $h = 4\tau$ (24,576 s while $\tau = 0$); 0 once σ exceeds 31	Core `vitality`, `_decay`	§9.2
E12	Depth bucket	$b = \min(15, \lfloor \log_{10}(d/10^{15}) \rfloor)$	Core `depthBucket`	§9.2
E13	Eviction hurdle	$2^{b(d_{\text{new}})}(1 + 0.25\,\mathbf{1}_{\text{bridge}}) > \Psi_{\text{worst}} + \lfloor \Psi_{\text{worst}}/4 \rfloor$	Hub `_canInsert`	§9.3
E14	Split gate	keep the split iff $\text{out}_{\text{split}} \ge \text{out}_{\text{single}} \cdot (1 + 25/10^{6})$, $\text{out}_{\text{single}}$ the best single leg at full size among every survivor of the believability band	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
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	$f_{\text{eff}} = p + f_{\text{key}} - \lfloor p\, f_{\text{key}}/10^{6} \rfloor$ if static, p the manager's protocol fee in the swap's direction; 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 $0 < f_{\text{decl}} \le 100$ 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

#	NAME	EQUATION	DEFINED IN	§
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 around the centre whose width each venue earns from its fee and its measured error (`_inBand`)	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 $\text{isBridge}(\text{tokenOut})$	Core `feeCalendar` , `paysOnInput` ; Router `_execute`	§13.1
E27	Emission closed form	$E(t) = 0$ for $t \leq s$; $R_b\tau + (S - (S \gg p)) + (R_0 \gg p)(\tau - pP)$ for $s < t < s + 4P$; $B + S - (S \gg 4)$ from $s + 4P$; $\tau = t - s, p = \lfloor \tau/P \rfloor, R_b = \lfloor B/L \rfloor, R_0 = \lfloor (S/2)/P \rfloor$	Staking `_emittedAt` (L1665–1675)	§25.3
E28	Emission window	$r = \min(\lfloor (E(t') - E(t_\ell)) \min(W, W_{\min})/W_{\min} \rfloor,\ RR)$; $A \mathrel{+}= \lfloor r \cdot 10^{18}/W \rfloor$	Staking `_windowReward` , `_updateGlobal` (L1620–1650)	§25.3
E29	Settlement	$\text{due}_u = (\lfloor w_u A/10^{18} \rfloor - c_u)^{+} - \text{excess}_u$, then $c_u \leftarrow \lfloor w_u A/10^{18} \rfloor$	Staking `_settleEmission` , `_settlePendingRewards` (L1699, L2141–2152)	§25.3
E30	Lock boost and weights	$b(\ell) = 10^4 + \lfloor 750\ell/365 \rfloor + \lfloor 250\ell^2/365^2 \rfloor$ bps, ℓ the lock in force, capped at 2,555 days; $w_u = \lfloor (s_u - d_u)^{+} b/10^4 \rfloor$; $w_u^{\text{p}} = \lfloor s_u b/10^4 \rfloor$ iff $d_u = 0$	Staking `boostByDays` , `_computeBoost` (L544–553, L1980–1985)	§25.3
E31	Interest curve	$r = 100 + \lfloor 500U/10^{18} \rfloor$ for $U \leq 0.8 \cdot 10^{18}$, else $500 + \lfloor 72{,}500(U - 0.8 \cdot 10^{18})/10^{18} \rfloor$ bps a year, with U capped at 10^{18} (ceiling 15,000 bps)	Staking `_interestRate` (L2074–2080) → `simulateRate` (L2623–2627)	§25.4
E32	Interest index and slice	$\delta = \lfloor \lfloor 10^{18} r/10^4 \rfloor \Delta t/Y \rfloor, \sigma = \lfloor D\delta/10^{18} \rfloor$; if $\sigma > S$, $\delta = \lfloor S \cdot 10^{18}/D \rfloor$ and σ recomputed	Staking `_interestSlice` , `_updateInterestIndex` (L1818–1890)	§25.4
E33	Slice split	$\kappa_r = \lfloor 300\sigma/10^4 \rfloor$ to the reserve; $\eta = \lfloor (\sigma - \kappa_r) \min(W_{\text{p}}, W_{\min})/W_{\min} \rfloor$ to pure stakers; the rest to the reserve	Staking `_updateInterestIndex` , `_dampPure` (L1818–1855, L1901–1903)	§25.4
E34	Interest attribution	$i_u = \lfloor d_u(I - I_u)/10^{18} \rfloor$, charged to s_u; any excess over s_u is a loss to E37	Staking `_accrueInterestFor` (L1905–1954)	§25.4
E35	Conservation guard	$\lvert \Delta(B + D + BD) - \Delta(S + RR + PR + \Pi) \rvert \leq 10^{10}$	Staking `conserves` (L467–505)	§25.5

#	NAME	EQUATION	DEFINED IN	§
E36	Owed and solvency	$\text{owed} = (S + RR + PR - D)^{+} + \Pi$; hard breach iff $B + 10^{10} < \text{owed}$	Staking `_owed`, `_hardBreach`, `isSolvent` (L2096, L2184, L2206)	§25.5
E37	Loss waterfall	$\kappa = \min(\lambda, PR)$; $PR \mathrel{-}= \kappa$; $BD \mathrel{+}= \lambda - \kappa$	Staking L1252–1257, L844–849, L1938–1946, `syncEquity` (L748–751)	§25.5
E38	Recapitalisation	$\rho = \min(BD, PR)$, $\gamma = \min(a, BD - \rho)$; $BD \mathrel{-}= \rho + \gamma$	Staking `absorbBadDebt` (L711–720)	§25.5
E39	Liquidation	liquidatable iff $d_u > 0 \wedge (s_u = 0 \vee 100 d_u \geq 95 s_u)$; $\text{seized} = \min(d_u + \lfloor 500 d_u / 10^4 \rfloor, s_u)$; loss $(d_u - s_u)^{+}$	Staking `_executeLiquidation`, `_isLiquidatable` (L1225–1288)	§25.5
E40	Emergency haircut	$x_0 = (s_u - d_u)^{+}$, $E' = E_{+} - e_u + x_0$; $x = \lfloor x_0 \,\text{pot} / E' \rfloor$ if $\text{pot} < E'$, else x_0; $x \leq \text{pot}$	Staking `emergencyWithdraw` (L821–835)	§25.5
E41	Borrowing ceilings	$d_u + a \leq \lfloor 50 (s_u - d_u)^{+} / 100 \rfloor$ and $\lfloor (D + a) 10^{18} / S \rfloor \leq 0.75 \cdot 10^{18}$	Staking `borrow`, `_ltvCap` (L974–980, L2305–2306)	§25.5
E42	Maintenance budget	$c(n) = \min(1 + \lfloor n/50 \rfloor + \lfloor (t - t_m)/900\ \text{s} \rfloor,\ 10,\ n)$ per registry	Staking `_windowBudget` (L1299–1308)	§25.5
E43	Walk gas window	$W = \min(\lfloor 63\,(S - s - 20{,}000)/64 \rfloor,\ \text{WALK_CAP})$, $S = \text{SOLVE_GAS} - 1{,}200$, s the gas the search spent; the solve stops if it entered on less than S or if $s + 20{,}000 \geq S$	Solver `_walkOnItsOwnBudget` (L324, L339–341)	§12.4
E44	Pool depth	$d = \min(\hat{x}_0, \hat{x}_1)$, each side normalised to 18 decimals; x_i the reserves for reserve families; $x_0 = \lfloor L Q_{96} / \sqrt{P} \rfloor$, $x_1 = \lfloor L \sqrt{P} / Q_{96} \rfloor$ for concentrated ones, $d = 0$ if $\sqrt{P} = 0$; the d of E12	Core `shortSide18`, `depthFromL18` (L1165, L1185)	§4.2
E45	Leg impact	reserve family: $\iota = \min(\lceil 10^4 x / (r_{\text{in}} + x) \rceil,\ 10^4)$, 10^4 if $r_{\text{in}} = 0$; concentrated: $\iota = (10^4 - \rho)^{+}$, ρ the realised output over the spot-price output in bps (10^4 if either is zero); 50 bps for every other leg and for a concentrated leg that quotes zero	Core `impactV2Bps`, `impactV3FromOut`, `legQuoteImpact` (L3412, L3469, L3084)	§11.5
E46	Route impact	$\delta = \lfloor \sum_h \sum_{\ell \in h} \lfloor \min(\iota_\ell, 10^4)\, n_h\, a_\ell / A_h \rfloor / N \rfloor$, n_h the legs of hop h, $A_h = \sum_{\ell \in h} a_\ell$, N the route's declared legs; the δ of E15	Core `wImp`, `routeFloorBps` (L3381, L3392); Router `_execute`	§11.5
E47	Per-leg floor	$\text{got}_\ell \geq \lceil 8{,}000\, \text{bound}_\ell / 10^4 \rceil$, else the swap reverts (`RouterE(5)`); $\text{bound}_\ell = \max(\lfloor e_\ell\, x / a_\ell \rfloor,\ \lfloor \lfloor q_\ell\, x / a^q_\ell \rfloor \cdot 5{,}000 / 10^4 \rfloor)$, x the amount the leg actually spent, e_ℓ, a_ℓ the caller's attested output and input, q_ℓ, a^q_ℓ the quote measured in-frame and its input, each term 0 when either of its pair is 0; on a reserve-family leg whose pool received $\text{took} < x$, $\text{bound} \leftarrow \lceil \text{bound} \cdot \text{took} / x \rceil$; no check while $\text{bound} = 0$	Router `_execScaled` (L1788–1791, L1839–1843, L1886, L1894)	§11.3

#	NAME	EQUATION	DEFINED IN	§
E48	Output net of input fees	for each hop h in order that pays on its input (E26): $\text{net} \leftarrow \lfloor \text{net}\,(b_h - f_h)/b_h \rfloor$ if $b_h > f_h$, else 0; $f_h = \lceil 28\,b_h/10^4 \rceil$, b_h the hop's planned input, net starting from the route's gross output; the frame in which the Solver attests `singleOutFloor` and the Quoter rescales it	Core `netOfInputFees`, `paysOnInput` (L461, L451); Solver L1911; Quoter L876	§11.5
E49	Band membership	$p_i = \lfloor r_i 10^6/(10^6 - f_i) \rfloor$ (r_i if f_i is unknown); $\kappa(f,e) = (1-f)(1-\min(e,1))$ in WAD, rounded down; row i is kept iff $p_i\,\kappa_i\,\kappa_* \,(1-f_*) \le p_*$ and (f_i unknown or $p_i \ge p_*\,\kappa_i\,\kappa_*^o$), with $\kappa_i = \kappa(f_i, e_i)$ ($f_i = 0$ if unknown), $\kappa_* = \kappa(f_*, e_*)$, $\kappa_*^o = \kappa(f_*, e_*^o)$, e_*^o the base's error bound at the hop's size; every product rounded down; a base whose fee is unknown keeps every live row	Math `_inBand`, `_keep`, `_prices` (L511, L543, L552)	§10.3
E50	Probe size	$\pi = \lfloor m/100 \rfloor$, or m when that is 0, m the hop's input; each candidate is quoted once at π for the band and the weights of E25	Solver `_buildHop` (L673–674)	§10.2
E51	Median weight	the tokenOut side the venue holds at its own price, in 18 decimals: the out reserve for reserve families; $\lfloor L\sqrt{P}/Q_{96} \rfloor$ or $\lfloor LQ_{96}/\sqrt{P} \rfloor$, the side paid out, for concentrated books and V4; for a Gate wrapper its depthWad, times $\hat{o}/\hat{\imath}$ (this quote's output over its input, 18 decimals) when $\hat{o} > \hat{\imath}$; the Solver caps a quoting reserve or concentrated row at its measured tokenOut balance and reads 0 as 1; the d_i of E25	Core `_quoteArms` (L1932–1936, L1961, L1997, L2014), `_v4Quote` (L2185), `outSide18` (L1201); Solver `_buildHop` (L745, L784)	§10.3
E52	Pool clock	on an anchored tick with $t_{\text{last}} \neq 0$ and $\Delta t = t - t_{\text{last}} > 0$: $\tau \leftarrow \tau + \lceil (\Delta t - \tau)/8 \rceil$ if $\Delta t \ge \tau$, else $\tau - \lceil (\tau - \Delta t)/8 \rceil$, from $\tau = 24{,}576/4 = 6{,}144$ s while $\tau = 0$; any other tick leaves τ as it was; the τ of E9 and E11	Core `tickSlot` (L3286–3305), `TAU_EMA_SHIFT` (L2925)	§9.2

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 editions, INV-18 and INV-22 brought to the code of this edition; INV-19–INV-22 were added in 2.3; INV-23–INV-31 are the staking engine's (§25), enforced in `src/BlazePhoenixStaking.sol` of the staking repository at `9958d5f`, with line numbers re-derived for that commit. B.1 is the numbered list; B.2 is the catalogue of the stateful `invariant_*` campaigns of both repositories, with the mutant that is on record as killing each.

B.1 The numbered invariants

#	INVARIANT	ENFORCEMENT
INV-1	The Router holds no user funds between transactions: the only balance it keeps at rest is a fee share a token refused to deliver, booked to its treasury slot (`owed`) and pulled by `claimFee`.	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

#	INVARIANT	ENFORCEMENT
INV-3	No oracle: no external price feed is an input to admission, ordering, splitting, flooring or fees.	No feed call exists in the contracts
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 `_rankedPools`
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)

#	INVARIANT	ENFORCEMENT
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
INV-18	No hop carries more than five legs and no route more than two hops, so no route carries more than ten legs; the bound is the product `MAX_HOPS` × `MAX_LEGS_PER_HOP` and nothing else, with one producer for the Router, the Quoter and the Solver.	Core `MAX_HOPS` = 2 (`BlazePhoenixCore.sol:136`), `MAX_LEGS_PER_HOP` = 5 (:149); Router `_execute` → `RouterE(3)` ; Quoter `_hookRefusedIn` ; the Solver builds every stage on the same constant
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 Hub, Solver, Router and Quoter ship under the project's 24,000-byte gate on the release profile (runtime 23,360, 21,253, 23,710 and 19,760 bytes at `969bcc6`); the Core, Math, Gate, the wrapper and the eight adapters are measured by the same step, and all 16 artefacts are under the 24,576-byte wall (§16.3). The default and release profiles carry the same compiler options.	`DeployedSizeGate` ; the CI size guard (fails at 24,000 bytes of runtime); `profile_parity.py` (§15.1: the suite itself is not run on the release profile in CI)
INV-23	On every staking entry point that wears `conserves` , balance plus debt plus recorded bad debt changes by the same amount as stake plus both reserves plus distributed-but-unpaid rewards, within 1e10 wei (E35).	Staking `conserves` , L467–505 → `Staking__InvariantBreached`
INV-24	The emission calendar closes exactly at `TOTAL_REWARDS` : `BASE_REWARDS + S - (S >> EMISSION_PERIODS) = TOTAL_REWARDS` , with remainder zero (E27); cumulative funding never exceeds `TOTAL_REWARDS` ; no emission window pays more than `rewardReserve` holds (E28).	`_emittedAt` L1665–1675; `fundEmission` L564 → `Staking__CapExceeded` ; `_updateGlobal` L1620
INV-25	Every realised loss — a liquidation shortfall, an under-water emergency exit, the `syncEquity` write-off of an under-water position in an emergency, uncollectible interest — draws on `protocolReserve` first; only the uncovered rest is added to `totalBadDebt` (E37).	Staking L1252–1257, L844–849, L748–751, L1938–1946

#	INVARIANT	ENFORCEMENT
INV-26	While the balance plus 1e10 is below `owed`, no deposit, borrow, withdrawal, reward or pure-yield claim, reserve withdrawal or emergency cancellation executes (E36).	`_hardBreach` gates L898, L968, L1040, L1069, L1084, L595, L686 → `Staking__InvariantBreached`; at L1791 a pure-yield settlement is credited to `_rewardDeferred` instead of paid
INV-27	`absorbBadDebt` moves no BZPX out, pays the caller nothing, and lowers `totalBadDebt` by at most its current value: the reserve first, the caller's amount capped at the uncovered rest (E38).	`absorbBadDebt` L711–720
INV-28	The totals of emission weight, pure-yield weight and positive equity are written only by `_applyBoost`, which replaces a position's tracked term with its recomputed one; a lock earns boost only while it is in force (E30).	`_applyBoost` L2020–2028; `_effectiveLockDays` L1972–1975
INV-29	A borrow executes only if the new debt is at most half of the position's net stake before it and the book's utilisation after it is at most 75 % (E41).	`borrow` L974 → `Staking__LTVExceeded`; L978–980 → `Staking__UtilTooHigh`
INV-30	An emergency exit pays at most the position's net equity and at most the contract's balance, scaled by pot/E′ when the balance is short of the tracked equity (E40).	`emergencyWithdraw` L821–835
INV-31	No withdrawal, pure-yield claim or lock change executes within `MIN_DEPOSIT_BLOCKS` = 10 blocks of the position's last deposit.	L1031, L1083, L1112 → `Staking__FlashLoanProtection`

B.2 The catalogue of stateful invariants

The DEX repository at `969bcc6` has 54 `invariant_*` functions in 18 files (the README says 54); the staking repository at `9958d5f` has 22 in 6 files. The tables list each by file, line and name. The column *Numbered invariant* is filled only where B.1 already cites the campaign by name. The column *Mutant on record* is filled only from the mutant measurement of 2026-09-05 in `docs/assurance/ASSURANCE.md` §4l (fifteen mutants, each run against every invariant then in the repository, 40 in 14 campaigns; eleven noticed on the first run; three of the other four gained, in their campaigns, the action they lacked and then killed their mutants; the fourth, the post-fee `userMinOut` check halved, survives by design, since no campaign universe reaches that guard and two unit mutants watch it instead). An invariant with a dash has no mutant named for it in that section: that is a statement about the record, not a finding that none exists. The curated mutation guard as a whole has 1,895 entries (§15.2).

DEX campaigns.

FILE (IN `test/`)	LINE	INVARIANT	NUMBERED INVARIANT	MUTANT ON RECORD
`AtqGateLifecycle.invariant.t.sol`	209	`invariant_AtqGateLifecycleMirror`	—	—

FILE (IN `test/`)	LINE	INVARIANT	NUMBERED INVARIANT	MUTANT ON RECORD
`AtqHubRegistry.invariant.t.sol`	186	`invariant_AtqHubRegistryMirror`	—	—
`AtqLedgerContext.invariant.t.sol`	647	`invariant_AtqRouterStateMachine`	—	—
`AtqPoolClock.invariant.t.sol`	218	`invariant_AtqPoolClockLaws`	—	—
`BlazePhoenixRouter.invariant.t.sol`	402	`invariant_RouterHoldsNothing`	INV-1	the final transfer pays one wei less
`BlazePhoenixRouter.invariant.t.sol`	414	`invariant_FeeNeverExceedsProtocolMax`	—	the protocol fee doubles
`BlazePhoenixRouter.invariant.t.sol`	424	`invariant_FeeNeverEscapes`	—	the input-side fee is never charged
`BlazePhoenixRouter.invariant.t.sol`	435	`invariant_SettledSwapEmitsExactlyOneFee`	INV-20	—
`BlazePhoenixRouter.invariant.t.sol`	441	`invariant_DeliveredNeverBelowTheProtocolFloor`	—	the protocol floor is halved (`FLOOR-half`)
`BlazePhoenixRouter.invariant.t.sol`	447	`invariant_TwoHopFeeShapeFollowsTheRule`	—	—
`BlazePhoenixRouter.invariant.t.sol`	451	`invariant_FeeIsChargedOnExactlyOneSide`	—	the fee is charged on both sides
`BlazePhoenixRouter.invariant.t.sol`	462	`invariant_DeliveredNeverBelowUserMinOut`	—	—
`DirtyWorld.invariant.t.sol`	825	`invariant_DirtyWorld`	—	—
`FullStackLoop.invariant.t.sol`	394	`invariant_TokenConservation`	—	30 % of every fee goes to a dead address
`FullStackLoop.invariant.t.sol`	412	`invariant_RouterHoldsNothing`	INV-1	the final transfer pays one wei less
`FullStackLoop.invariant.t.sol`	423	`invariant_RegistryNeverExceedsMaxSlots`	INV-8	—

FILE (IN `test/`)	LINE	INVARIANT	NUMBERED INVARIANT	MUTANT ON RECORD
`FullStackLoop.invariant.t.sol`	437	`invariant_ActiveEntriesResolveToRealPools`	—	—
`FullStackLoop.invariant.t.sol`	456	`invariant_SettledPlansOnlyTouchAttestedPools`	—	—
`HubInvariantFromV1.t.sol`	200	`invariant_neverExceedsMaxSlots`	INV-8	`MAX_SLOTS` becomes 17
`HubInvariantFromV1.t.sol`	207	`invariant_ActiveEntriesTradeThePair`	INV-19	`recordSwap` 's pair-proof is removed (`INVARIANT-AIMED PAIRPROOF-off`)
`HubInvariantFromV1.t.sol`	216	`invariant_activePoolsResolve`	—	—
`OnlyControlOneBody.t.sol`	391	`invariant_OnlyControl_ADoorOpensOnlyToTheModelsLiveAdmin`	—	—
`QuoterPreviewDelivery.invariant.t.sol`	167	`invariant_accountingNeverDrifts`	—	—
`QuoterPreviewDelivery.invariant.t.sol`	173	`invariant_encodeAgreesWithPreview`	—	—
`QuoterPreviewDelivery.invariant.t.sol`	179	`invariant_canExecuteSettlesAtItsFloor`	—	—
`QuoterPreviewDelivery.invariant.t.sol`	185	`invariant_exactOutIsAFloorTheRouterHonours`	—	—
`RouterAdversarialInvariantFromV1.t.sol`	319	`invariant_conservationA`	—	30 % of every fee goes to a dead address
`RouterAdversarialInvariantFromV1.t.sol`	322	`invariant_conservationB`	—	30 % of every fee goes to a dead address
`RouterAdversarialInvariantFromV1.t.sol`	325	`invariant_conservationH`	—	30 % of every fee goes to a dead address
`RouterAdversarialInvariantFromV1.t.sol`	328	`invariant_routerHoldsNothing`	INV-1	the final transfer pays one wei less

FILE (IN `test/`)	LINE	INVARIANT	NUMBERED INVARIANT	MUTANT ON RECORD
`RouterAdversarialMultiHopFromV1.t.sol`	123	`invariant_conservationA`	—	30 % of every fee goes to a dead address
`RouterAdversarialMultiHopFromV1.t.sol`	124	`invariant_conservationB`	—	30 % of every fee goes to a dead address
`RouterAdversarialMultiHopFromV1.t.sol`	125	`invariant_conservationC`	—	30 % of every fee goes to a dead address
`RouterAdversarialMultiHopFromV1.t.sol`	128	`invariant_routerHoldsNothing`	INV-1	the final transfer pays one wei less
`RouterAdversarialV4FromV1.t.sol`	281	`invariant_conservationA`	—	30 % of every fee goes to a dead address
`RouterAdversarialV4FromV1.t.sol`	284	`invariant_conservationB`	—	30 % of every fee goes to a dead address
`RouterAdversarialV4FromV1.t.sol`	287	`invariant_routerHoldsNothing`	INV-1	the final transfer pays one wei less
`RouterAdversarialV4FromV1.t.sol`	291	`invariant_reentrancyBlocked`	—	the reentrancy lock no longer refuses
`RouterMultiHopInvariantFromV1.t.sol`	131	`invariant_routerHoldsNothing`	INV-1	the final transfer pays one wei less
`RouterMultiHopInvariantFromV1.t.sol`	143	`invariant_conservationA`	—	30 % of every fee goes to a dead address
`RouterMultiHopInvariantFromV1.t.sol`	144	`invariant_conservationB`	—	30 % of every fee goes to a dead address
`RouterMultiHopInvariantFromV1.t.sol`	145	`invariant_conservationC`	—	30 % of every fee goes to a dead address
`RouterSplitInvariantFromV1.t.sol`	121	`invariant_routerHoldsNothing`	INV-1	the final transfer pays one wei less
`RouterSplitInvariantFromV1.t.sol`	132	`invariant_conservationA`	—	30 % of every fee goes to a dead address
`RouterSplitInvariantFromV1.t.sol`	133	`invariant_conservationB`	—	30 % of every fee goes to a dead address
`RouterStatefulConservationFromV1.t.sol`	108	`invariant_conservationA`	—	30 % of every fee goes to a dead address
`RouterStatefulConservationFromV1.t.sol`	115	`invariant_conservationB`	—	30 % of every fee goes to a dead address
`SqSameAnswer.t.sol`	596	`invariant_Sq_BothSolversAnswerTheSameBytes`	—	—

FILE (IN `test/`)	LINE	INVARIANT	NUMBERED INVARIANT	MUTANT ON RECORD
`StrandedRegime.invariant.t.sol`	306	`invariant_StrandedMoneyIsNeverSwept`	—	the input-residual or bridge-residual sweep ignores its baseline (`SWEEP-bridge-no-base`)
`StrandedRegime.invariant.t.sol`	315	`invariant_HoldsNothingBeyondTheSeed`	INV-1	the final transfer pays one wei less
`StrandedRegime.invariant.t.sol`	322	`invariant_DivergentLegNeverSettles`	—	the leg-pair guard admits a leg on the wrong pair (`LEG-pair-and`)
`StrandedRegime.invariant.t.sol`	330	`invariant_PausedRouterNeverSettles`	—	`whenLive` no longer checks `paused`
`StrandedRegime.invariant.t.sol`	335	`invariant_LedgerConservationA`	—	30 % of every fee goes to a dead address
`StrandedRegime.invariant.t.sol`	343	`invariant_LedgerConservationT`	—	—

Staking campaigns (file in `test/`).

FILE	LINE	INVARIANT	MUTANT ON RECORD
`BlazePhoenixStaking.t.sol`	447	`invariant_solvent`	—
`BlazePhoenixStaking.t.sol`	451	`invariant_conservationBit`	—
`BlazePhoenixStaking.t.sol`	455	`invariant_boostBounded`	—
`BlazePhoenixStaking.t.sol`	461	`invariant_backingCoversOwed`	—
`BlazePhoenixStaking.t.sol`	472	`invariant_everyLiveCommitmentIsTracked`	—
`BlazePhoenixStaking.t.sol`	482	`invariant_lockRegistryHasNoDuplicates`	—

FILE	LINE	INVARIANT	MUTANT ON RECORD
BlazePhoenixStaking.t.sol	493	invariant_staleBoostExcessIsBounded	—
InvariantsFullSurface.t.sol	278	invariant_INV16_conservationIdentityHoldsExactly	—
InvariantsFullSurface.t.sol	307	invariant_INV16_backingCoversOwed	—
InvariantsFullSurface.t.sol	338	invariant_totalStakedShortfallIsExplainedByUncollectedInterest	—
InvariantsFullSurface.t.sol	386	invariant_STAKEATTR_stakeLedgerIsExact	—
InvariantsFullSurface.t.sol	401	invariant_INV18_lapsedLockPaysBaseBoost	—
InvariantsFullSurface.t.sol	421	invariant_INV20_breakerCannotTripWhileSound	—
InvariantsFullSurface.t.sol	426	invariant_INV19_debtNeverExceedsStake	—
InvariantsFullSurface.t.sol	441	invariant_EQUITYFRESH_accumulatorIsTheSumOfStoredPositiveEquity	—
InvariantsFullSurface.t.sol	553	invariant_EQUITYFRESH_EMG_accumulatorIsTheStoredSumThroughEverySyncAndExit	—
assurance/EmissionScheduleMatrix.t.sol	519	invariant_DistributedNeverExceedsTheScheduleWhichNeverExceedsTotalRewards	—
assurance/EmissionScheduleMatrix.t.sol	527	invariant_ReservePlusDistributedIsAlwaysWhatWasFunded	—
atq/AtqEmergencySurface.invariant.t.sol	223	invariant_AtqEmergencySurfaceKeepsTheBooksUpToTheSimulatedLoss	—

FILE	LINE	INVARIANT	MUTANT ON RE-CORD
`attack/EmissionShareInvariant.t.sol`	127	`invariant_ATK_INV_paidLeDistributedLeCurveLeCap`	—
`attack/EmissionShareInvariant.t.sol`	142	`invariant_ATK_INV_noActorAboveItsShare`	—
`hunt/ExactAttributionInvariant.t.sol`	104	`invariant_campaignActuallyGeneratesDebt`	—

The mutant column is a mapping by invariant from the measurement of 2026-09-05, and it covers the DEX only: a dash means that no mutant is recorded against that invariant, not that none exists. The 22 staking invariants carry no such mapping.

B.3 Symbolic proofs (Halmos)

Fifteen Halmos checks of the DEX repository report `[PASS]` in the CI run at commit `6387350`. Halmos executes each check symbolically over the compiled bytecode, for every input that satisfies the check's stated preconditions. The eight in `CoreFormalGateSpec` and `EffV4FeeFormalSpec` are the CI gate. The column *Code under check* names the function the check calls, with its line in `src/` at `6387350`.

FILE (IN `TEST/FORMAL/`)	LINE	CHECK	PROPERTY	CODE UNDER CHECK
`CoreFormalGateSpec.t.sol`	30	`check_ironFloorBps_boundsUnconditional`	For every impact, leg count and σ: $8{,}000 \le$ `ironFloorBps` $\le 9{,}600$	Core `ironFloor` (:3402)
`CoreFormalGateSpec.t.sol`	40	`check_impactV2Bps_neverExceedsBPS`	For every amount and reserve, the V2 impact is at most 10,000 bps	Core `impactV2Bps` 3412)
`CoreFormalGateSpec.t.sol`	53	`check_outV3_failClosedGuards`	A zero amount, zero liquidity, zero price or a fee of at least 10^6 quotes 0	Core `outV3` 1431)
`EffV4FeeFormalSpec.t.sol`	22	`check_staticKeyIsTruthWithoutProtocolFee`	A static key with no protocol fee in the swap's direction returns the key fee verbatim; the `slot0` LP fee is ignored	Core `effV4` (:2244)
`EffV4FeeFormalSpec.t.sol`	33	`check_staticKeyComposesProtocolFee`	With a protocol fee p in the swap's direction, a static key composes $p + f - \lfloor pf/10^6 \rfloor$: never below the key fee, never above the key fee plus p, and never in the unquotable band for any LP fee the manager admits	Core `effV4` (:2244)

FILE (IN TEST/FORMAL/)	LINE	CHECK	PROPERTY	CODE UNDER CHECK
EffV4FeeFormalSpec.t.sol	46	check_dynamicProtoFeeFailsClosed	A dynamic-fee key with any non-zero protocol fee resolves to at least 10^6 (unquotable)	Core effV4 (:2244)
EffV4FeeFormalSpec.t.sol	53	check_dynamicUsesMeasuredLpFee	A live dynamic-fee pool (no protocol fee) prices from the slot0 LP fee	Core effV4 (:2244)
EffV4FeeFormalSpec.t.sol	59	check_sentinelNeverSurvivesForLivePool	The dynamic-fee sentinel is never the resolved fee of a live pool	Core effV4 (:2244)
CompositionFormalSpec.t.sol	32	check_depthIsShortSide	For non-zero liquidity and price, the token-denominated depth of a concentrated-liquidity pool is at most both virtual reserves, $\lfloor LQ_{96}/\sqrt{P} \rfloor$ and $\lfloor L\sqrt{P}/Q_{96} \rfloor$	Core depthFromL (:123 mirrored in t
CompositionFormalSpec.t.sol	44	check_depthEqualsLiqAtUnitPrice	At $\sqrt{P} = Q_{96}$ the depth equals L	Core depthFromL (:123 mirrored in t
SolverBandFeeDustFormalSpec.t.sol	78	check_lemmaMulmodFits	Lemma L0: for $a, b < 2^{128}$, $\mathrm{mulmod}(a, b, 2^{256}-1) = ab$, so the high word of mulDiv is 0 and its 512-bit branch is not taken	integer lemm the guard of mulDiv (:4
SolverBandFeeDustFormalSpec.t.sol	86	check_lemmaHopFeeIsCeiling	Lemma CEIL: for $b < 2^{128}$, the code's fee satisfies $28b \le 10^4 \cdot \mathrm{fee} < 28b + 10^4$, that is $\mathrm{fee} = \lceil 28b/10^4 \rceil$	Core mulDi (:531)
SolverBandFeeDustFormalSpec.t.sol	121	check_bandNoBaseAndDeadRow	With no base row ($b \ge n$) the band keeps nothing; a row priced 0 is never kept	Math _inBa (BlazePhoenixMath 511)
SolverBandFeeDustFormalSpec.t.sol	149	check_hopFeeRoundsUpAndNetMatches	Given L0 and CEIL (the two checks above), for every $b < 2^{128}$: $b > 0 \Rightarrow \mathrm{fee} > 0$; $b \ge 2 \Rightarrow \mathrm{fee} < b$; the Core's one-hop net of the input fee equals $\lfloor \mathrm{base}\,(b - \mathrm{fee})/b \rfloor$	Core mulDi (:531), netO putFees (:
SolverBandFeeDustFormalSpec.t.sol	182	check_dustProbeV2FloorsToZero	The Solver's dust probe ($m/100$, or m when that is 0) is at least 19 wei for $m \ge 1{,}900$; a 1-wei constant-product quote at 30 bps and price at most 1 is 0	Core outV2 1398)

B.4 Properties of the compiled artefact

The DEX repository's `assurance metrics` workflow (`.github/workflows/assurance.yml`, line 57) runs `.github/scripts/assurance/bytecode_invariants.py` over the runtime bytecode of every `BlazePhoenix*` contract the build produces, and the step fails if any of the seven properties below does not hold. The script reads opcodes by walking the code with PUSH data skipped, so a constant is never read as an instruction. Lines are those of the script at `6387350`, where the workflow passes.

LINE	PROPERTY
43	No runtime contains `SELFDESTRUCT`.
44	No runtime contains `CALLCODE`.
45	No runtime contains `ORIGIN`: `tx.origin` is no authority anywhere in the system.
67–74	`BlazePhoenixMath`, the library the other contracts reach by `DELEGATECALL`, contains none of `SSTORE`, `TSTORE`, `CREATE`, `CALL`, `DELEGATECALL` and `CREATE2`.
155–166	Every function selector the ABI declares appears as a constant in the runtime dispatcher: no declared function is unreachable.
167–191	Every refusal `revert E(n)` in a contract's source file has its code n as a push immediate in that contract's runtime, so no refusal the source contains was removed by the optimiser.
213–214	The Router's runtime contains `TLOAD` and `TSTORE`: the reentrancy lock and the fee-on-transfer probes are transient.

B.5 Specified properties

This table states each property and where it is specified; the guarantees of this paper are the invariants of B.1 and the entries of B.2–B.4. The Certora rules are in `certora/specs/` of the DEX repository at `6387350`, each run by the configuration of the same name in `certora/conf/` (`effv4fee.conf`, `floorsfee.conf`, `router-control.conf`, `router-swap.conf`); the Halmos checks are in `test/formal/` of the DEX repository at `6387350` and of the staking repository at `9958d5f`.

SPECIFIED IN	LINE	RULE OR CHECK	PROPERTY
`EffV4Fee.spec`	27	`staticKeyIsTruthWithoutProtocolFee`	A static key with no protocol fee in th swap's direction returns the key fee ve batim.
`EffV4Fee.spec`	35	`staticKeyComposesProtocolFee`	With a protocol fee p, a static key reso to $r = p + f - \lfloor pf/10^6 \rfloor$, with $f \le r \le f + p$ and $r \le 10^6$.
`EffV4Fee.spec`	47	`dynamicWithProtoFeeFailsClosed`	A dynamic-fee key with any protocol resolves to the unquotable value (≥ 1
`EffV4Fee.spec`	55	`dynamicUsesMeasuredLpFee`	A live dynamic-fee pool prices from th `slot0` LP fee.

SPECIFIED IN	LINE	RULE OR CHECK	PROPERTY
`EffV4Fee.spec`	60	`sentinelNeverSurvivesForLivePool`	The dynamic-fee sentinel is never the solved fee unless `slot0` reports it.
`FloorsFee.spec`	25	`constantsArePinned`	BPS = 10,000; protocol fee 28 bps; leg f 8,000 bps; floor base 9,600 bps; hard m um loss 2,000 bps.
`FloorsFee.spec`	37	`protocolFeeIsCeilOf28Bps`	$\text{fee} = \lceil 28\,\text{base}/10^4 \rceil$.
`FloorsFee.spec`	45	`protocolFeePaysAndNeverSwallows`	$\text{base} > 0 \Rightarrow \text{fee} > 0$; $\text{base} \geq 2 \Rightarrow \text{fee} < \text{base}$.
`FloorsFee.spec`	53	`protocolFeeMonotone`	A larger base never gives a smaller fee
`FloorsFee.spec`	61	`legFloorIsCeil80`	The leg floor is $\lceil 0.8 \cdot \text{bound} \rceil$ and n above the bound.
`FloorsFee.spec`	71	`legFloorMonotone`	A larger bound never lowers the leg fl
`FloorsFee.spec`	79	`hopRuleCollapsesToLegFloorAtN1`	For one leg, the hop rule is equivalent the leg floor.
`FloorsFee.spec`	86	`hopSlackNeverExceedsOneAverageLeg`	n times the slack is at most 20 % of th bound.
`FloorsFee.spec`	96	`moreLegsNeverLoosensHopSlack`	More legs only tighten the slack.
`FloorsFee.spec`	103	`hopBarMonotoneInBound`	A larger hop bound never lowers the h minus the slack.
`FloorsFee.spec`	113	`ironFloorBounds`	$8{,}000 \leq \texttt{ironFloorBpsShv} \leq 9$
`FloorsFee.spec`	123	`ironFloorMonotoneInShaves`	The floor is non-increasing in impact, shave and σ.
`FloorsFee.spec`	130	`ironFloorCountFormBounds`	The leg-count form of the floor lies in $[8{,}000, 9{,}600]$.
`RouterControl.spec`	53	`controlGateOnSetters`	A control-only setter succeeds only fr the admin, and only before renunciati
`RouterControl.spec`	62	`configOnlyByControl`	Configuration changes only through c trol; a treasury slot also through its ho
`RouterControl.spec`	79	`renounceIsForever`	Once `controlRenounced` is true i stays true.
`RouterControl.spec`	87	`renounceRefusedWhilePaused`	Renunciation reverts while the Router paused.
`RouterControl.spec`	95	`neverPausedAndRenounced` (invariant)	Paused and renounced are never both
`RouterControl.spec`	101	`queueSetsFortyEightHourEta`	A queued rescue becomes executable 4 hours after it is queued.

SPECIFIED IN	LINE	RULE OR CHECK	PROPERTY
`RouterControl.spec`	109	`rescueTimelockAndSingleUse`	A rescue executes only if queued, only its delay, and only once.
`RouterControl.spec`	118	`rescueNeverTakesOwedFees`	A rescue never takes the fees owed to treasuries.
`RouterSwap.spec`	58	`zeroMinOutRefused`	A zero `userMinOut` is refused at ev door.
`RouterSwap.spec`	76	`amountInAboveU128Refused`	An `amountIn` above $2^{128} - 1$ is re fused.
`RouterSwap.spec`	94	`pausedBlocksEverySwapDoor`	While the Router is paused, every swa door reverts.
`RouterSwap.spec`	104	`deliveryAtLeastMinOutOrRevert`	A one- or two-hop swap pays at least `userMinOut` or reverts.
`RouterSwap.spec`	118	`routerHoldsNothing`	After a swap the Router holds none of swapped tokens beyond the fees it ow
`CompositionFormalSpec.t.sol`	62	`check_hopCollapsesToLegFloorAtN1`	For one leg, the hop rule is equivalent the leg floor.
`CompositionFormalSpec.t.sol`	71	`check_slackNeverExceedsOneAverageLeg`	n times the slack never exceeds one fi the hop bound.
`CompositionFormalSpec.t.sol`	80	`check_moreLegsNeverLoosensBudget`	More legs only tighten the slack.
`CoreFormalSpec.t.sol`	33	`check_outV2_neverReachesReserveOut`	For a non-zero input and non-zero res in the `uint112` range and a fee of a most 1,000 bps, the constant-product c is strictly below the output reserve.
`CoreFormalSpec.t.sol`	49	`check_outV3_zeroForOne_boundedByVirtualReserve`	The in-tick output never exceeds the v reserve $L\sqrt{P}/Q_{96}$.
`CoreFormalSpec.t.sol`	63	`check_mulDiv_matchesExactWhenNoOverflow`	For a, b in the `uint128` range and $d > 0$, `mulDiv` equals $\lfloor ab/d \rfloor$.
`MathLibFormalSpec.t.sol` (staking)	40	`check_SafeIsZeroWhenAnyOperandIsZero`	`mulDivSafe` returns 0 when any op and is 0.
`MathLibFormalSpec.t.sol` (staking)	48	`check_MulDivReturnsForNonZeroDenominatorInWordRegime`	For `uint128` operands and a non-ze denominator, `mulDiv` returns.
`MathLibFormalSpec.t.sol` (staking)	61	`check_AgreesWithPlainDivisionInWordRegime`	For `uint128` operands, `mulDiv` e plain integer division.

SPECIFIED IN	LINE	RULE OR CHECK	PROPERTY
`MathLibFormalSpec.t.sol` (staking)	71	`check_SafeAgreesWithUnsafeInWordRegime`	For `uint128` operands, `mulDivSa` and `mulDiv` agree.
`MathLibFormalSpec.t.sol` (staking)	78	`check_DivideByOwnFactorIsIdentity`	$\mathrm{mulDiv}(a, b, b) = a$ for $b \neq 0$.

Appendix C · Reproduction commands

Every number in this paper that comes from the DEX repository is reproduced by one of the commands in the first block, run from a clean checkout of `Blaze-Phoenix-Dex` at `969bcc6` (at `6387350` for the figures the text dates at that commit) with Foundry and Python 3 installed; fork suites need an archive RPC key in `DRPC_KEY`. The numbers of the staking engine (§25) are reproduced by the second block, from a clean checkout of `Blaze-Phoenix-Staking` at `9958d5f`. Where a figure of the previous edition was measured at an earlier commit and not run again at `969bcc6`, the text of its section says so, and the command here is the way to run it again.

```
forge test --no-match-path 'test/fork/**'                        # the suite, release
settings (§15.1)
forge test --match-path 'test/fork/**'                           # the fork lane (§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 (§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
forge test --match-contract RegimeCoverageT3Test -vv              # strength 3
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'        # the metamorphic rela-
tions (§15.7)
FOUNDRY_PROFILE=nver1 forge build --skip '*.t.sol' && FOUNDRY_PROFILE=nver2 forge build
--skip '*.t.sol' && FOUNDRY_PROFILE=nver3 forge build --skip '*.t.sol'
NVERSION_LANE=1 forge test --match-path 'test/nversion/*'         # the binaries agree
(§15.8)
forge test --match-test invariant_ --fuzz-seed 0xb1a2ef00         # the stateful cam-
paigns (§15.3, Appendix B.2)
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
(§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
(§17.2)
forge test --match-path test/fork/DiscoveryColdWarm.t.sol -vv     # cold and warm legs,
no gas (§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          # the projection-dis-
tance table (§15.9)
python3 .github/scripts/assurance/profile_parity.py               # the default and re-
lease profiles carry the same options (INV-22)
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 (Appendix D)
```

```
# Blaze-Phoenix-Staking at 9958d5f
forge test                                                    # the staking suite
(§25.7)
forge test --match-test invariant_                          # the invariant campaigns (Ap-
pendix B.2)
forge test --match-path test/assurance/EmissionScheduleMatrix.t.sol   # the emission cal-
endar (E27, E28, INV-24)
```

The curated mutation guard (`mutants.py`) lists 1,895 mutants at `969bcc6`, counted from the list in the script; a mutant is a small change, usually to a single line (1,692 of the 1,895 replace exactly one non-blank line) and to the contracts (1,865 of the 1,895 edit a file under `src/`), that a named test must turn red. 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.

Appendix D · The shared-quantity catalogue

The project's rule is *one question, one producer*: a quantity that more than one place produces or consumes either collapses to a single producer, guarded in CI against copies, or keeps its producers and carries a named test that asserts they agree. The register that enumerates them (`SHARED_QUANTITIES.md` in the DEX repository at `969bcc6`) holds 131 rows (134 at `6387350`: the three added since are the route's floor rate `BPC.routeFloorBps`, the encode door's `finalHopQuote` read through `Quoter._askLeg`, and the funnel's horizon `FUNNEL_STRETCHES`). This appendix gives the public form of it: each quantity, the question it answers, and its producer. The register's status and pin columns are not reproduced here, because a status is a claim about the day it was written; read them in the repository at the commit you are checking. The staking repository keeps its own register (37 rows at `9958d5f`; §25.7).

Floors and fees.

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`LEG_FLOOR_BPS`	"How much may a single leg legitimately lose?"	`Core`
hop commitment (`Σ leg.amountIn`)	"How much does this hop commit to trade?"	`Router._legSum`
fee count per settlement	"Did this swap pay the protocol — once on an anchored route, once per hop on an exhausted one?"	`Router._payFee` counts into a transient slot, `Router._execute` reads it once at settlement: zero → `RouterE(15)`, more than one on an anchored route → `RouterE(16)`.
`owed` (a fee share the payee refused, seal 2026-10-21)	"Whose is the fee share a push could not deliver, and may a swap see it?"	`Router._push` books it
Router balance after settlement (`RouterBalanceAfterSettlement`, header R1)	"What may the Router still hold when a swap ends?"	every payout of `Router._execute`

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`PROTOCOL_FEE_BPS`	"How much does the protocol take?"	`Router._chargeHopFee` ×2, `Quoter._pack` ×1, `Quoter.previewPlanExact` ×2
`effV2Fee` / `quoteV3Fee`	"What fee does this pool actually charge?"	`Core` only
`ironFloorBps` **impact input**	"What is this route's price impact?"	`Solver` ×2
`expectedOut` of a single-tick leg (the per-leg attestation)	"What can this leg be promised to deliver?" - and, from the walk (2026-09-24) until S2 (2026-10-11), the same answer to "which venue delivers more?". **Since S2 (V4 fragmentation, reported by Seavia Resources) the two questions have two figures:** the promise stays the sixteen-stretch walk, a floor an outsider can only lower (safe for a promise); the ranking is `rank` from `Core.v4WalkRank`.	produced by `Core.universalQuote`'s V4 arm through `Core.v4WalkOut`, the pool's own swap walked over its initialized ticks.
`v4WalkOut` (what a V4 pool pays for an input)	"What does this V4 pool pay for this amount?"	`Core.v4WalkOut`
promise of a capacity-held concentrated leg (`MAX_CONC_DRAIN_BPS`, 2026-10-22)	"What is a leg promised when the capacity bound holds its promise but not its input?"	`Solver._capacity`, at both commit sites
`v3PaysOnBudget` (what a V3-shaped pool pays for an input, on a budget of reads; DEX-01 / 3331, 2026-10-27)	"What may a concentrated leg be promised, and what does the Router quote for it in frame?"	`Core.v3PaysOnBudget`
the price a concentrated range lands on for an exact input (`_nextSqrtFromInput`; 1 wei of rounding, 2026-11-10)	"Where does this range's price land when this input enters it?"	`Core._nextSqrtFromInput`
`_v4Quote` on a dynamic-fee key whose hook runs beforeSwap (the LP fee the swap charges)	"What LP fee does this V4 swap pay?"	Two producers that cannot agree: the quote reads slot0's `lpFee` through extsload (`Core.effV4Fee`), the PoolManager charges the fee the hook RETURNS from beforeSwap on a dynamic key (v4-core `Pool.swap`: the override wins over slot0).
`sqrtPriceAtTick` (the price at a tick)	"Where does a tick's price lie?"	`Core.sqrtPriceAtTick` alone - the TickMath table, each constant derived from its definition
`_askWord` (what one pool's answer may cost a quote)	"How much gas, memory and trust may one answer take from the planner?"	`Core._askWord` - a one-word ask with `GAS_CAP` of gas and one word copied, `ok` only for a full word

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`ASK_GAS_FLOOR` (the gas a one-word ask must hold; aero-nova, 2026-10-26)	"May a starved ask answer 'no'?"	`Core._askWord` alone, before its call: on less than `ASK_GAS_FLOOR` = 104 587
`getV3Fee` budget (`CL_FEE_COST` , `CL_FEE_WINDOW` , `CL_FEE_SPENT` , `CL_FEE_GAS` , `CL_FEE_TRUST` , `CL_FEE_FLOOR` ; aero-nova, 2026-10-26)	"How much gas may one concentrated pool's fee() take, and when is its answer believed?"	`Core.getV3Fee` alone, on its own budget instead of the shared `GAS_CAP`
`SOLVE_GAS` (what the in-transaction search may spend)	"Does the plan `swapBestExactIn` executes equal the plan `findBestRoutePlan` returns off-chain, and what happens when the search needs more gas than it is given?"	`Router.swapBestExactIn` calls `Solver.findBestRoutePlan` with a fixed `SOLVE_GAS`
`hookPaused` (a listed hook whose code moved since its pin)	"May a leg through this hook execute?"	`Hub.hookPaused`
`hookAltersDeltas` (a hook with its own accounting)	"May a leg through this hook execute?"	`Core.hookAltersDeltas`
`_hookRefusedIn` (the Router's refusal of a leg, at the exact doors)	"Will the Router refuse this route before it moves a token (RouterE(3) shape, RouterE(9) hook, RouterE(11) key, RouterE(8) native side or no engine, RouterE(19) Gate wrapper not live)?"	`Quoter._hookRefusedIn` is the one producer the preview asks: `_pack` withholds `canExecute` with it, and `_exact` (both `previewPlanExact` and `previewRouteExact`) sets the exact scalar to zero with it.
`deadline` at the encode doors	"Will the Router refuse this call before it moves a token (RouterE(4))?"	`Router._swap`
`effectiveMinOut` at the encode doors (the floor inside the bytes)	"Will the Router refuse this call's delivery (RouterE(5))?"	`Router`
`exactOut` with one pool twice in a hop	"What will the Router deliver for this route in this block?"	`Quoter._exact` prices every leg against the pool before the swap
`_dropDust` (the funnel's dust rule: a row's measured depth against the rest of its pair)	"Which rows of a pair does the funnel quote?"	`Solver._dropDust` , in `Solver._rankedPools` on every pair with two rows or more: each row is measured by `Core.hopMass18`
`hopMass18` (a hop's mass: the tokenOut a row can pay, in 18 decimals)	"How much of the bought token can this row pay?"	Two producers, one formula
best single leg at full size (the min-split gate's reference)	"Is this split worth its legs?"	`Solver._buildHop` : every survivor of the band, including the ones the funnel cut dropped

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`MAX_ROUTE_IMPACT_BPS` (the route's ceiling on one leg's measured impact)	"Which leg is a destroyer?"	`Solver._assembleRouteMulti`, the one judge, refusing the route when any leg's impact (`_legImpactBps` : V3 through its fee, V2 by its input against its reserve) reaches it.
band vote (whose mass may choose the band's base)	"Who votes the depth-weighted median?"	`Math._depthWeightedMedian`, masked by `Math._mutedVotes` : a live pair- or concentrated-shaped candidate whose row a swap wrote
one price order per hop (`_byPrice`, M11, wave 5.2 gas, 2026-10-28; `_byRate`, by rate, until MED the same day)	"In which order do a hop's live prices stand, for its plain median and for every weighted median of the hop?"	`Math._byPrice` : one stable insertion sort of the live rows
a venue's price as the band reads it (`Math._prices`, MED, 2026-10-28)	"Which price does the band compare, and with which fee and error bound?"	`Math._prices` : p = r * 1e6 / (1e6 - f), r the probe's rate after fee (the Solver's `rates[i]`), f the fee that probe was charged and e its error bound, both from `Core.universalProbe` and packed by the Solver into one word, `fe[i]` , the fee in bits 0-23 and e above.
a row's error bounds `err` and `errOrder` (`Core.universalProbe`, ROUTEQ, 2026-11-13; two numbers)	"How far can this row's price sit from the spot at the probe, and how far can the base's fill of the WHOLE order fall under it?"	`Core.universalProbe(c, probe, bar, wantFee, orderIn)` returns both, in WAD, with the rounding terms rnd = (3 WAD + 2 probe) / rank + 1 at the probe in both.
first stage of a bridge route (`_firstStage`, M10, wave 5.2 gas, 2026-10-28)	"What does the first stage tIn -> bs[i] build, in each topology that opens on bs[i]?"	`Solver._buildHop` builds it once per solve and `Solver._firstStage` keeps a copy in the pair's `_candsOf` entry
remainder seat (the leg that takes what the shares leave over)	"Who absorbs the rounding and the input the capacity clamp frees?"	`Math._seatTheSplit` : the deepest survivor by measured weight, ties to the lowest address - never the funnel's psi order
`sumPsi` (the split's denominator, S)	"Over what total are a split's shares floored, so that the remainder is never negative?"	`Math.select` returns it as the sum of psis[0..n)
a leg's place in the split (`_placed`, EVERY LEG PAYS FOR ITS PLACE, 2026-10-28)	"Does this leg pay for its place in the split?"	`Solver._placed` : in a split of three legs or more, the leg whose probe paid least
`QuoteDeltas` (the Quoter's dry-run payload)	"Is this revert data a quote?"	produced by the `Quoter` fallback
`_inputTaxed` (the Quoter's input probe, at the exact doors)	"Will the pool receive less of this leg's input token than the Router pays it (RouterE(13))?"	`Router._v3Callback` pays a concentrated pool what it demands and refuses the leg when the pool's measured delta is short

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
the net output the exact door promises (`exactOut` of `previewPlanExact` / `previewRouteExact` , 2026-11-13)	"What does the Router deliver for this route, in this block, after the fee the calendar names?"	`Quoter._fundedOut` prices the route as the Router funds it
the fee walk (`Quoter._walk` , 2026-11-14)	"What does each hop of this route receive and yield once the calendar's input fee is taken where the Router takes it?"	`Quoter._walk` , the one producer for the exact pass of `_exact`
`finalHopQuote` (the anchor of the protocol floor)	"What does the floor anchor on when the last hop measured nothing?"	`Router.swapExactIn` , hop loop: the last hop's in-frame quote, else its attested figure, else - when every leg of it is blind - the hop's declared `expectedOut` .
`_cutByWeight` (which band survivors the leg budget keeps)	"Which pool does the funnel cut drop at equal weight?"	`Math._cutByWeight` : heaviest measured weight, then the better marginal rate
`ExecutionProof.floorUsed` against `realized`	"Which floor did this settlement meet, in the units it delivered?"	`Router.swapExactIn` : `protocolFloorOut` is checked against `outCap` , the gross the venues declared, before the output-side fee and any transfer tax
concentrated holdings cap (`registryDepth18` , concentrated arm)	"How much mass does a concentrated pool hold?"	`Core.registryDepth18` through `Core.heldMass18`
`reservesHeld` / `heldBy` (does a reserve-shaped pool hold what it reports)	"May anything believe this pair's reserves?"	`Core.heldBy`
`v4HeldShare` (a singleton book's vote: its depth scaled by the share of its virtual output side it holds)	"With how much depth does this V4 book vote in the band?"	`Core.v4HeldShare`
`legQuoteImpact` (the in-frame quote and impact of a pair-shaped or concentrated leg; K2, 2026-10-21)	"What does this leg quote, and at what impact, as the frame executes it?"	`Core.legQuoteImpact`
`_derivedByFactory` (admission requires proof of origin)	"May this pool take a seat through the swap door?"	`Hub.discoverFor`
`_scanV4` (what a V4-derive row lists for a pair)	"Which V4 pools of this pair does discovery list?"	`Hub._scanV4` , called by `Hub.discoverFor` for each `MODE_V4_DERIVE` row: after the custody check
`_pmHoldsNone` (does the PoolManager hold any of a token)	"Can any V4 pool of this pair pay this token out?"	`Hub._pmHoldsNone` , asked by `Hub._scanV4` before its first probe for each side that is neither a bridge nor WETH: the manager's unlock flag
per-leg floor bound (`_execScaled` coverage gate)	"What does a leg's floor rest on when its attestation is low?"	`Router._execScaled` : `bound = max(attested, qs * MIN_QUOTE_COVERAGE_BPS / BPS)` where `qs` is the in-frame quote rescaled to the executed amount

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`MIN_QUOTE_COVERAGE_BPS` (5 000)	"Under what share of the in-frame quote may a leg's bound never rest?"	`Core`
`residIn` (the route input the legs did not spend)	"Whose is the input that no leg consumed?"	two producers in `Router._execute`, one per route shape: the post-loop sweep (`tokenIn ≠ tokenOut`) and, for a CYCLIC route (`tokenIn = tokenOut`), the refund at the start of hop 1 - after the loop residual and output are the same token and can no longer be told apart.
a split's dust remainder (`m - Σ leg.amountIn` on one hop; FindingAnchorDonation, 2026-11-13)	"How much of an order may a split leave to no leg, and whose is it?"	`Solver._split` : a leg whose quote at its share is 0 is dropped and its share is not handed on, so `committedIn` < m
`legOutRaw` (the token a leg delivers into)	"Which token does this leg produce?"	produced once per leg by `Router._legTokens`
`declared` (the ceiling of a leg's measured delta)	"What did the venue say it paid the Router for this leg?"	produced per leg kind: `Router._execPairAmt` returns `outAmt` , the output the Router asks of the pair
`outCap` (the ceiling of the swap's output delta)	"How much of the tokenOut the Router gained since `toutStart` is this swap's output?"	produced in `Router._execute` , from zero: a leg that makes tokenOut adds the `declared` its `_execScaled` returns
`owned` (the bridge a later hop may spend)	"How much of the Router's balance of hop h>=1's input is this swap's?"	produced by `Router._owned` for two consumers: `_chargeHopFee` (the fee base of hop h>=1) and `_hopScaleImpact-AndQuote` (`realIn` , the input hop h>=1 scales its legs to).

Registry state (the Monoslot).

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
registry `fee`	"What fee should the planner assume for this pool?"	written by `Hub._register` , read by `Hub._readPoolInfo` → Solver/Quoter
registry `hooks`	"Does this pool have a hook?"	written by the two doors that admit one, `Hub.addV4` and `Hub.seedPool` , each after `_admitOnRegistration`
registry `tier` (Monoslot bits [47:40])	"Did a curator vouch for this row?"	built by each registration door in `Hub._born(kind, fee, trusted)` and handed to `Hub._register` inside the born slot: `addV4` and `seedPool`
`depthWad` (via `depthFrom-L18`)	"How deep is this pool?"	`Core` ×3 - the two arms of the quote body
`bridged` (Monoslot bit 7)	"Is this pool anchored on a routable bridge?"	written frozen at registration

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`2_500`, the bridge term (+25 %)	"How much does a bridged pair add to a row's score?"	two producers, pinned by parity: `Core.psi` scores each incumbent
`stable`	"Is this a stable-curve pool?"	the pool's own `stable()`, read once by `Core.solidlyStable` at the registration door
pool depth **source**	"Who says how deep this pool is?"	`Router._recordHits` reads `getReserves` from the pool
reserve-shaped leg promise vs `Holdings`	"Can this pair pay what its leg promises?"	the promise is `outV2` /Solidly on the pair's own `getReserves`
`getReserves` word width (`Reserve`)	"Is this word a reserve at all?"	`Core.getReserves`, read by every reserve-shaped quote and by `Router._recordHits`
`provenShape` (a non-V4 row's family)	"What family is this pool?"	`Core.provenShape`
`solidlyAskOut` (what a Solidly pair is asked to pay)	"How much does a Solidly leg deliver, and so how much may it promise?"	`Core.solidlyAskOut`
`_solY` exit (the replicated stable curve's output reserve, behind `outSolidlyStable`)	"Does the replicated stable curve leave K_post >= K_pre in contract arithmetic?"	`Core._solY`
`single` (the min-split gate's reference)	"What would this order get from the best single pool?"	`Solver._buildHop` alone: every survivor of the band and the funnel is quoted at full size by `_singleLeg`
`decIn1` / `decOther1` (a hop's two token decimals, +1 encoded in `QuoteCtx`)	"How many decimals do this hop's tokens have?"	read once per pair per solve by `Solver._rankedPools`
`QuoteCtx` of a candidate (`xs` , a hop build's contexts)	"In which context does the Core quote this candidate?"	one producer, `Solver._ctx`, called once per candidate per build by `_buildHop`
the G7 harness Hub's V4 derive row (`G7Deploy._v4Wire` ; G7 robinhood midtail-1, 2026-11-14)	"Does the campaign's Hub run the V4 derive scan on this chain?"	`G7Deploy._v4Wire(PM, WETH, stable)` is the one producer in the harness: `hub.v4PoolManager` and one `addFactory` row with `KIND_V4` and `MODE_V4_DERIVE`, on every chain with a PoolManager (Optimism, Arbitrum, Robinhood since 2026-11-14; Base already).

Identity and keys.

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`computeV4PoolId`	"Which V4 pool is this leg talking about?"	`Core`
`v4LegCurrencies` (the native side of a V4 leg)	"Which side of this native V4 leg is the chain's coin?"	`Core.v4LegCurrencies`, by orientation
`nativeMapVerified` in the preview (#52, seal 2026-10-21)	"Will the Router accept this native V4 leg's key?"	`Core.nativeMapVerified`
`keyOf`	"What is this pool's registry key?"	`Core.pairKey`, the one producer
`KINDS_ROUTABLE` / `KINDS_EXECUTABLE` / `KINDS_PAIR_PROOF`	"Which kinds may enter / execute / must prove their pair?"	`Hub`
`hookAltersDeltas` bit map	"Does this hook return deltas?"	`Core`
`MAX_BRIDGES` ↔ Solver arms	"How many bridges does the planner actually walk?"	`Hub` ×7, `Solver` ×2
`MAX_LEGS_PER_HOP` (5; finding PIN-01)	"How many legs fit in one hop?"	`Core:145`, the one producer since 2026-11-06: the Router refuses a wider hop with `RouterE(3)`
`MAX_HOPS` (2)	"How many hops fit in one route?"	`Core:132`, the one producer since 2026-11-06
`singleOutFloor` (the floor the Solver attests, in the frame the Router compares it in)	"Against which input was this floor sized, and against which is it compared?"	`Solver._assembleRouteMulti`, one producer for every route since 2026-11-11
`feeHop` (the hop the protocol fee anchors on)	"On which hop is the fee taken, and in which bridge?"	`Core.feeCalendar`, the one producer since 2026-10-27
Layer 2 scope (hookless before hooked)	"Over what span may no hookless leg follow a hooked one?"	`Router._execute`
`Volume`	"How much really went through this pool?" (`amtIn`, `amtOut`)	emitted by `Hub.recordSwap` from the amounts the Router passes

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`legTook` / `legOut` (what one leg's venue took in and delivered, per leg of the route)	"How much did THIS venue actually receive, and how much did it actually pay?"	produced once per leg by `Router._execScaled` : `legOut` is the Router's own tokenOut balance delta over the leg, bounded by the venue's declared payment (DEX-11).
`v4EntryOf` (V4 row → V4Entry)	"Where is this V4 row's tickSpacing?"	written by every V4 door
V4 row identity at the seed door (the row's pool vs the id its V4Entry derives)	"Which pool does this V4 row measure?"	`Hub.seedPool` recovers the tier on the hookless id
`factoryDeployer` (the attested Algebra origin)	"Which CREATE2 origin do this factory's mode-5 fee-0 probes derive from?"	written by `Hub.addFactory` from `Core.resolvePoolDeployer` at admission
`factories[]` row ↔ `factoryCodehash` / `factoryDeployer` (keyed by address)	"Is this factory already admitted, and which row is it?"	the array holds the rows, the mappings are keyed by address
`factories[row].initHash` (the CREATE2 derivation input of a live row)	"Which address does discovery derive for this factory?"	written by `Hub.addFactory` on both arms
`finalHopQuote` (the figure the output floor is a fraction of)	"What should this hop have delivered?"	produced in `Router._execute` from `hopQuote` : the sum over the hop's legs of each leg's in-frame quote, or its attested figure when the frame could not price it (a Gate leg, a native leg with no in-frame quote; since 2026-10-23, so a hop with no in-frame quote at all reads `hopBound` as before).
`SELF` (the account the Router code executes as)	"Whose balances does this swap measure and move?"	two producers of one answer: `address(this)` , which under an EIP-7702 delegation is the delegating EOA, and the deployment address, fixed in the code as the immutable `SELF` .
`v3Salt` (the V3 salt of (t0, t1, fee))	"Under which salt does a V3-salt factory deploy the pool of (t0, t1, fee)?"	`Core.v3Salt`
`liveAdapter` (Gate: which adapter may run for a family)	"Which adapter may trade for family f, right now?"	`Gate.liveAdapter` is the one producer
`venuePool` (Gate wrapper)	"Which venue pool does this wrapper trade on?"	written once, at `Gate.create` , from the family adapter's `resolve`
`KIND_EXTERNAL` fail-closed (kind 9 where the Core has no reader)	"What does the Core read for a kind with no reader?"	two Core sites: `Core.provenShape` (refuses), `Core.registryDepth18` (0).
`KINDS_V3_SALT`	"Which kinds may sit in the V3 salt slot (mode 5)?"	`Hub.addFactory` only

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
kind-9 doors (where a Gate leg may execute)	"Through which doors may a kind-9 leg execute?"	every door: the Router's execution dispatch serves kind 9 at the three explicit doors and at the best door's self-call `selfExecutePrePulled`, always behind `Core.gateWrapperLive` (row `kind-9 identity`).
kind-9 identity (is this pool a Gate wrapper?)	"Is the pool of a kind-9 leg the admitted Gate's wrapper for this hop and fee?"	`gateWrapperLive` in the Core, the one producer
Gate quote (price of a kind-9 leg)	"What is a Gate leg worth?"	`Core.gateQuote`, the one producer since 2026-10-22: identity first
`_requireWrapper` (Gate adapters, adapters-1021)	"May this caller make the adapter move tokens, and which tokens?"	`ExtAdapterBase._requireWrapper` asks the Gate's own registry
`_quote` (Curve NG)	"What does this NG leg pay for amtIn?"	one private body read by `quote`
`_quote` (Curve crypto NG)	"What does this Twocrypto-NG / Tricrypto-NG leg pay for amtIn?"	one private body read by `quote`
`_out` (Liquidity Book)	"What does this Liquidity Book leg pay for amtIn?"	one private body read by `quote`
`_out` (Algebra Integral)	"What does this Algebra Integral leg pay for amtIn?"	one private body read by `quote`
`_requireHookPinned` (Gate adapters)	"May this pool's hooks run under the adapter?"	`ExtAdapterBase._requireHookPinned`, the one producer since 2026-11-14
`_hooksOk` (Liquidity Book)	"May this pair's hooks run under the adapter?"	one function over the pair's `getLBHooksParameters`: read at `resolve`
`_pluginOk` (Algebra Integral)	"May this venue's plugin run under the adapter?"	one function over the plugin config of the venue's state read: BEFORE_SWAP and DYNAMIC_FEE are refused always.
`_requireHookPinned` (Infinity CL, Infinity Bin)	"May this pool's hooks run under the adapter?"	`ExtAdapterBase._requireHookPinned`

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
adapter `depth18`	"How deep is this external leg?"	`ExtAdapterBase._depth18`, the one adapter producer: the shorter side of the balances the venue itself reports, each normalised by its token's decimals, through `Core.shortSide18`, the registry's own depth of a pair (`registryDepth18` 's reserves arm, `depthFromL18`), so one mass reads one depth on both doors.
adapter `depth18` (Liquidity Book form)	"How deep is this Liquidity Book leg?"	`ExtAdapterBase._depth18`, the row above, fed by a fifth form, by reserves: the pair's `getReserves`, what it reports its LPs own
adapter `depth18` (Algebra Integral form)	"How deep is this Algebra Integral leg?"	`ExtAdapterBase._depth18`, the row two above, fed by the reserves form: the venue's `getReserves`, what it books, never its balanceOf, so a donation adds no depth.
Gate use (`vitalSlot`, a wrapper's swap count)	"How used is this Gate wrapper?"	`Wrapper.swap`, the one writer, after the custody checks: Core's `tickSlot` then `stampTs`, in the Hub's slot layout, only when the swap is material on both sides of the venue pool (input >= the venue's balance of the input token >> `MATERIAL_SHIFT`, output >= its balance of the output token >> `MATERIAL_SHIFT`, output non-zero; each side in its own token).
Gate fitness (`vitalOf` : vitality, depth18, psi of a wrapper)	"How fit is this Gate wrapper for the registry?"	`Gate.vitalOf`, the one producer, relayed by `Quoter.gateVitalOf` behind `Core.gateWrapperLive` : zeros unless the Gate created the wrapper and its family is live.
kind-9 registry feedback (does a Gate leg reach `Hub.recordSwap`)	"Who records that a kind-9 leg was used?"	`Wrapper.swap` alone
`wrappersOf` (the Gate's per-pair list; `MAX_WRAPPERS_PER_PAIR` = Core's `GATE_MAX_WRAPPERS` = 4)	"Which wrappers may the automatic door seat for this pair?"	`Gate.create` is the one writer
`gatePsi` (a wrapper's score in the Solver's funnel)	"How does the funnel order a kind-9 row?"	`Gate.vitalOf` 's psi, read by `Core.gatePsi` in the Solver's ordering loop in place of the Hub's psi, which is 0 for a pool with no Hub row
pool clock (`tau` , bits [255:224] of a registry row; the Gate's `vitalSlot` alike)	"How often does this pool trade, on its own clock?"	Core's `tickSlot`, the one writer of a row's clock: the count, tau and `lastUpdateTs`
learned V4 code (`v4CodeOf` , a token's V4 launch-tier hint)	"At which hookless V4 tier does this token trade?"	`Hub._writeV4Code`, the one body, reached from four doors: `claimV4` after `_register` admitted the row
`scratch` (where an assembly call or copy writes its answer: 0x00-0x3f)	"Where may a raw call put the words it reads?"	Solidity's scratch 0x00-0x3f, or memory the caller allocated
`MODES_READ_FEE` / `MODES_READ_SPACING` (which grid axes a factory mode reads)	"Do two cells of a row's grid ask its factory different questions?"	`Core.deriveAddress` / `_factoryLookup`

Transient state.

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`TSLOT_FOT`	"What net ratio (WAD, compounded) did this swap's taxed transfers keep, for the route floor? 0 = none seen"	`Router._noteFot`
`received` (what a door's pull delivered)	"How much `tokenIn` did this call's pull actually put on the Router?"	`Router._received`, one producer since 2026-10-28
transient slot numbering	"Which slot holds what?"	`Router`
`realIn` (hop h ≥ 1: `_owned(bal - kept, bridgeBase[h-1], hopGot - fee)`)	"How much of the bridge may this hop spend?"	`Router._hopScaleImpactAndQuote`

Measurement harness (tools/g7-gas-campaign).

QUANTITY	THE QUESTION IT ANSWERS	PRODUCER
`active_b0` (liquidity in range at the window start b0; G7 campaign, 2026-11-14)	"Can this concentrated pool fill a swap at b0?" (not "how much is in the pool?")	`build_spec.active_at` is the one producer: `liquidity()` of a V3-shaped pool

Appendix E · The venues

This appendix lists, in one place, every venue family the DEX supports as its README states it at `6387350`, and for each one how a pool is found, how it is priced and whether the price is a closed form the Core computes or an answer it delegates. The mode numbers are those of the resolution table in §7.1 and the kind numbers those of §4.1. A quote is advice to the caller in every row; the Router's execution floors read what arrived, never the quote (§4.4).

VENUE	KIND	HOW A POOL IS FOUND	HOW THE PRICE IS MEASURED	CLOSED FORM OR DELEGATED
Uniswap V2 and constant-product forks	0	factory call (mode 0), or CREATE2 arithmetic where the deployment is reproducible (mode 4); the pair proves its `token0` / `token1`	`getReserves`, after the pair is shown to hold what it declares (`heldBy`)	closed form `outV2` (§4.2)
Uniswap V3 and V3-shaped forks	1	factory call (mode 1), or CREATE2 arithmetic (mode 5); the pool proves its pair	`slot0` and the active liquidity for the ranking; the pool's own ticks walked for the promise (`v3PaysOnBudget`)	closed form for the ranking; the venue's own book for the promise
Uniswap V4	4	pool ids derived from public parameters and each proven live against the PoolManager's storage (mode 9); no address, no call	the tick book walked through `extsload`, 16 steps for the promise, up to 256 for the ranking	walked in the frame, no venue answer trusted (§5)

VENUE	KIND	HOW A POOL IS FOUND	HOW THE PRICE IS MEASURED	CLOSED FORM OR DELEGATED
Uniswap V4, native ETH	8	as V4; never registered by a factory row	as V4	as V4
Algebra (dynamic fee)	6	CREATE2 arithmetic with the Algebra sentinel (mode 5, `fee == 0`, origin the factory's `poolDeployer()`), or a factory call (mode 1)	as kind 1, the live fee measured from `globalState`	closed form, fee measured
Solidly (stable and volatile)	5	factory call (mode 2) or CREATE2 arithmetic (mode 6); the pair proves its tokens	the pool's `getAmountOut` less one wei; the replicated curve is the fallback when the pool does not answer	ask-the-venue first; the replicated stable-curve solver as fallback (§4.2)
Balancer V2	9	a Gate wrapper at the address the Gate's creation code derives (the Hub's `MODE_CREATE2_V3` row); created by the curator	`gateQuote`: identity, then the wrapper's own `quote`, through the family's adapter	delegated to the adapter (`AdapterBalancerV2`)
Balancer V3	9	as Balancer V2	as Balancer V2	delegated (`AdapterBalancerV3`)
Curve StableSwap-NG	9	as Balancer V2	as Balancer V2	delegated (`AdapterCurveNG`)
Curve Two-crypto-NG and Tricrypto-NG	9	as Balancer V2	as Balancer V2	delegated (`AdapterCurveCryptoNG`)

Two rules apply to every row of kind 9. The Router measures each leg's balance delta and refuses one above what the wrapper declared it paid (RouterE(20)); the per-leg floor and the hop budget read that measured delta, the caller's minimum reads what reaches the recipient, and the route floors read the gross the venues declared (`outCap`) (§4.5). An adapter enters only after the `ADAPTER_DELAY` of 48 hours, with its code hash pinned, and a changed adapter stops every wrapper of its family at the next call.

Three further families stand behind the Gate on the same terms as Balancer V2 (kind 9, delegated to the family's adapter): Liquidity Book, LFJ v2.1 and v2.2 and Pharaoh DLMM (`AdapterLB`); Algebra Integral custom venues (`AdapterAlgebraIntegral`); PancakeSwap Infinity CL and Bin (`AdapterInfinityCL`, `AdapterInfinityBin`).

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, seven kinds in five families today (the seventh kind, external venues behind the Gate, is delegated, §4.5)

SYMBOL	NAME	ONE LINE
$\mathcal{D}$	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 % (8,000 bps), applied to the final hop's in-frame quote
Ψ	Vitality Field	venue quality computed from the Monoslot alone; ranks the candidate set and settles exact ties — it can order 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. Adapter — the per-family contract behind a Gate wrapper that speaks to the external venue; a new or replaced adapter takes effect only after the 48-hour `ADAPTER_DELAY` (§4.5). **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** — the no-arbitrage band around the depth-weighted median of the candidates' prices; a candidate outside it is not believed (§10.3). **Coverage gate** — a leg's attested bound raised to at least half the measured in-frame quote (`max(attested, quote × 5,000 / 10,000)`), never lowered. **Deployment (staking)** — one instance of the staking contract with its own reward reserve, protocol reserve and emission clock; the engine is deployed twice, and the two do not see each other (§25.8). **Exact Pass** — the Quoter's pool-exact preview: the venue's own swap run and rolled back by revert-extraction. **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. **Gate** — the contract that puts every external venue behind an immutable per-venue wrapper; a pair has at most four wrappers (§4.5). **Hermes plugin** — a plugin for the Hermes agent runtime that gives an agent four BlazePhoenix tools; it reads the user's node from the agent's secret store and asks the hosted API to prepare and decode a quote, checking both against the node's answer (§27.5). **Hop** — one token-to-token step of a route; a route has at most 2 hops. **Leg** — one pool's share of a hop; a hop has at most 5 legs, so a route has at most 10. **MCP server** — a tool server for AI agents speaking the Model Context Protocol; the local one runs on the user's machine over the SDK and the user's node, the hosted one prepares and decodes calls that the caller's node runs (§27.3, §27.4). **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: `renounceControl` writes a flag no code path can unset; the control tier ends, 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. **Wrapper** — the immutable per-

venue contract of the Gate: a V3-shaped `swap` and `uniswapV3SwapCallback` in front of a curated adapter whose code hash is pinned. **X-Ray** — the SDK's second entry point, which measures by simulation on a pinned block whether a token can be bought, sold and transferred at the tax the caller accepts; NOT_MEASURED is a status, not a pass (§27.2).

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 Curve family, which the DEX reaches through the Gate and does not replicate (§4.5).
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. Mitra. *BlazePhoenix-Dex: an on-chain DEX aggregator with measured routing*. Repository, `main` 969bcc6, 2026-11-16. https://github.com/blazephoenixxyz-crypto/Blaze-Phoenix-Dex — The canonical source for every claim about the DEX 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).
29. Mitra. *Blaze-Phoenix-Staking: the BZPX staking engine, Staking V4*. Repository, `main` 9958d5f, 2026-11-24. https://github.com/blazephoenixxyz-crypto/Blaze-Phoenix-Staking — The canonical source for every claim about the staking engine in §25; the second block of Appendix C reproduces its numbers from it.
30. Mitra. *BlazePhoenix SDK* (`@blazephoenix/sdk`). Repository, `main` 2d2fc39 (1.7.0), 2026-11-21; npm 1.6.0 from `bb41243` . MIT licence. https://github.com/blazephoenixxyz-crypto/SDK — The client, the Phoenix Check, the calldata verification and the X-Ray of §27.2.

31. Mitra. *BlazePhoenix MCP server* (`@blazephoenix/mcp`). Repository, `main` 3514804 (1.3.0), 2026-11-20. MIT licence. https://github.com/blazephoenixxyz-crypto/blazephoenix-mcp — The local MCP server of §27.3.
32. Mitra. *Blaze-Phoenix-API.* Repository, `main` 01c5468, 2026-11-20. MIT licence. https://github.com/blazephoenixxyz-crypto/Blaze-Phoenix-API — The hosted MCP endpoint and the REST API of §27.4.
33. Mitra. *BlazePhoenix Hermes plugin.* Repository, `main` cf38638 (1.0.0), 2026-11-21. MIT licence. https://github.com/blazephoenixxyz-crypto/blazephoenix-hermes-plugin — The Hermes plugin of §27.5.

How to cite

Mitra. *BlazePhoenix Technical Whitepaper: an on-chain liquidity aggregator whose quote and execution are one computation.* Version 2.4.0, 2026-11. Zenodo. DOI: to be assigned by Zenodo on deposit. BibTeX and other formats: https://blazephoenix.xyz/research/whitepaper

Earlier versions, each with its own record: version 2.3.1, 2026-11-16, DOI 10.5281/zenodo.24372939; version 2.3.0, 2026-09-24, DOI 10.5281/zenodo.23084091; concept DOI of that record, 10.5281/zenodo.23084090; concept DOI of the first deposit series (version 2.2), 10.5281/zenodo.22526574. A citation of the numbers in this paper cites 2.4.0 and, with it, the commits named in Appendix C: `969bcc6` for the DEX and `9958d5f` for the staking engine.