certora

Security Assessment
Final Report

Balancer

Balancer V3

December 2025 - January 2026

Prepared for Balancer

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Balancer V3	balancer-v3-monorepo	fd1c3712d8681e35caaf8a2691ca36d351c436ea	EVM
Balancer V3	balancer-v3-monorepo	PR 1601	EVM
Balancer V3	balancer-v3-monorepo	PR 1607	EVM

Project Overview

This document describes the manual code review findings for the Balancer V3 project. The work was undertaken from **December 1st 2025** to **January 26th 2026**.

All contracts in the balancer-v3-monorepo are considered in scope.

Notes on PRs (1601 and 1607) reviewed

These PRs are defense-in-depth improvements to fixed-point rounding and parameter validation, aimed at reducing security-relevant edge cases and misconfiguration risk. The original implementation was already sound and conservative, these changes preserve existing behavior, and primarily tighten correctness at numerical boundaries.

Protocol Overview

Balancer v3 is the successor of v2 and is a decentralized automated market maker (AMM) protocol built for EVM chains with a clear focus on fungible and yield-bearing liquidity.
Balancer v3's architecture focuses on simplicity, flexibility, and extensibility at its core. The v3 vault more formally defines the requirements of a custom pool, shifting core design patterns out of the pool and into the vault.

Balancer Pools are smart contracts that define how traders can swap between tokens on Balancer Protocol, and the architecture of Balancer Protocol empowers anyone to create custom pool types.

A major innovation in v3 is native Vault support for the ERC4626 standard. This allows boosted pools to have 100% capital efficiency. A boosted pool is simply a pool of any type containing ERC4626 wrapped tokens. Vault buffers and convenience functions in the Router allow users to interact with boosted pools as if they were standard pools containing only underlying tokens.

Threat model

In the following section, we define the threats and risks for automated market makers in general, and Balancer protocol in particular. For each we have determined how Balancer has mitigated these risks.

Scope & assets

Protocol components in scope

- Vault (centralized storage for tokens and accounting).
  Core of v3 design.
- Pool implementations / pool hooks (custom pool logic, swap math, fee logic, hooks for composing behaviour).
- ERC4626 buffers that swap assets in external vaults and maintain liquidity buffers.
- Router contracts: User interactions with the Vault, handles token transfers approvals and multi step operations.

High-value assets

- Token balances held in Vault (user LP funds, ERC4626 buffer liquidity).
- Protocol-owned tokens (protocol Fees)
- BPT tokens held by users

Primary actors & motivations

- External attacker: profit-seeking , exploit math bugs, precision issues, reentrancy, oracle manipulation, arbitrageable invariant flaws.
- Malicious or buggy integrator: third-party pool or hook deployer with risky logic.
- Insider/ops compromise: compromised private keys for deployer/guardian/relayer multisigs.
- MEV bots/front-running: sandwiching, oracle manipulation, liquidity-draining sequencing.

Risks and threats

1. Smart Contract Bugs in Vault

The Balancer V3 Vault is the central storage and accounting layer for all pools. Mismanagement or bugs in the vault could compromise all pools.
Risks:

1. Loss of funds due to incorrect accounting.
2. Unauthorized access or misuse of vault admin functions.
3. Token transfer failures leading to stuck liquidity.
4. Errors in fee accrual or distribution.

We have evaluated all access to admin functions, mathematical calculations in the vault and verified the proper updating of accounting. More details on the vault calculations and safeguards can be found in the section Vault calculations and safeguards.

2. Smart Contract Bugs in Core Pools

Core Balancer V3 pools manage asset swaps, liquidity provisioning, and fee accrual. Bugs in pool contracts could result in fund loss, incorrect swaps, or pool exploitation.
Risks:

1. Reentrancy attacks during swaps or liquidity changes.
2. Logic or rounding errors in swap or liquidity calculations.
3. Unexpected behaviour due to edge-case token transfers (e.g., non-standard ERC20s).
4. Permission misconfigurations in pool controllers or vaults.

We have manually audited the core pool math and swap/liquidity flows, and tested rounding and edge cases to reduce the likelihood of precision, invariant, and reentrancy-related issues. Detailed analysis of the pool calculations can be found in the section Pool calculations and safeguards.

3. Malicious or Unsafe Pools

Balancer V3 allows anyone to create pools, which can introduce risks if the pool is designed with malicious intent or unsafe configuration. These pools may contain poorly implemented logic, exotic tokens, or hidden mechanisms that put liquidity providers at risk.
Risks:

1. Hidden transfer fees or malicious token logic draining LP funds.
2. Pools with imbalanced weights or extreme parameters causing disproportionate impermanent loss.
3. Exploitable smart contract bugs in user-created pools.
4. Users being tricked into providing liquidity to unsafe or scam pools.

We have reviewed the Vault's pool registrations, interaction boundaries and tested key safety checks (permissions, hooks, and settlement constraints) to minimize the consequences of misconfigured or adversarial pool logic.
The vault has clear isolation between pools, making sure that potential issues in a pool cannot affect other pools in the vault. Balancer has implemented procedures to ensure that only verified and audited pools by trusted parties are added to the Balancer frontend.
Interacting with untrusted pools is at the user's own risk.

4. ERC4626 Wrapped Yield Tokens

Balancer V3 pools can wrap yield-bearing tokens using ERC4626 vaults to allow liquidity providers to earn additional yield. A liquidity buffer of vault shares and underlying tokens is stored in the vault to optimize swaps.
Risks:

1. Wrapper share price manipulation or desynchronization.
2. External protocol failure propagating into wrapped assets.
3. Buffer depletion causing reverts, forced slow-path withdrawals, or mispriced swaps.
4. Exploits in underlying yield strategies (front-running, sandwich attacks).
5. Token wrapping/unwrapping errors causing rounding loss or pool imbalance.
6. Liquidity manipulation impacting yield calculations or pool balances.
7. Contract upgrade risks if vaults are upgradeable.

We have manually reviewed the ERC4626 wrapping/unwrapping, buffer mechanics, buffer depletion, rounding, and rate-change scenarios to ensure the system behaves safely across share-price and ERC4626 liquidity edge cases.
The core protection is that wrap/unwrap is executed while enforcing initialization and pause checks, is non-reentrant, verifies the ERC4626's underlying asset matches expectations, and applies the same user limit semantics as swaps. To reduce rounding leakage and share-price manipulation edge cases around ERC4626 previews, the implementation applies conservative ±1 wei adjustments and enforces a minimum wrap amount so tiny amounts cannot be used to extract value through precision loss. If the buffer is insufficient, the Vault follows a controlled slow-path that explicitly rebalances the buffer as part of the operation rather than silently mispricing. Residual risks remain inherent to the external vault/strategy, but the integration is designed so these risks manifest as bounded outcomes (revert via limits or clean failure).

5. Router Contract Risks (User Interaction Layer)

Balancer V3 uses router contracts to simplify user interactions with the Vault, handling token approvals, deposit flows, swaps, batch operations, and multi-step transactions. These routers form a critical bridge between end-users and the Vault. Bugs, unsafe assumptions, or UX pitfalls in these contracts can lead to failed operations, stuck assets, or user-side losses.
Risks:

1. Stuck Tokens: Incorrect handling of token transfers or callbacks may result in tokens trapped in the router rather than reaching the Vault.
2. Approval Misuse: Users may grant overly broad or perpetual approvals, which could be misused by malicious contracts or compromised routers.
3. Unexpected Reverts: Complex multi-call routing may fail mid-execution, leaving users with partially executed operations or unexpected token states.
4. Slippage Misconfiguration: Routers rely on user-provided parameters, incorrect slippage limits may cause preventable transaction failures.
5. Execution Ordering Issues: Multi-operation routes may depend on assumptions about token balances or pool states that can change between calls.
6. Malformed Routes or Inputs: Incorrect path construction or unsupported token behaviors (rebasing, fee-on-transfer) can cause errors or value loss.
7. Front-End Misrouting: UI bugs or malicious interfaces could redirect user actions to unintended router contracts.

We have manually reviewed router flows for approvals, multi-call sequencing, token movement, and validated common failure modes to minimize risks of stuck tokens and unsafe parameter handling. Routers forward user intent to the Vault through `unlock`, and the Vault enforces correctness (limits, accounting, and state updates), so routers are not trusted for slippage safety. Malformed routes or unsupported token behaviors are expected to fail atomically because routing executes inside unlock, any invalid step (wrong pool/token relationship, unsupported token mechanics) causes a revert. Router hook entrypoints are additionally restricted to `onlyVault` and are `nonReentrant` so external callers cannot directly invoke hook logic. ETH handling is intentionally conservative due to the router only accepting ETH from WETH unwrapping and returning leftover ETH to the contract caller, reducing accidental stuck funds and making refund behavior explicit.

6. Token-Specific Risks

Balancer supports a wide range of ERC20 tokens, some of which have unusual behaviours (e.g., rebasing, fee-on-transfer).
Risks:

1. Rebasing tokens causing pool imbalance or incorrect accounting.
2. Fee-on-transfer tokens reducing liquidity or skewing balances.
3. Non-standard ERC20 behaviour causing transaction failures or stuck liquidity.

Though Balancer officially does not support non standard ERC20 tokens, we have reviewed integrations assuming non-standard ERC20 behavior and tested edge cases such as fee-on-transfer and unusual decimal/scaling behavior We have verified that for most unsupported token mechanics, there are no accounting issues in the vault. The Vault does not rely on “expected transfers” alone. Rather, it computes settlement credit from actual `balanceOf` deltas and caps credit to the caller’s hint to avoid miscounting pre-sent tokens.
For decimals and scaling oddities, the system operates on scaled values and applies rounding directions that favor the Vault, plus it enforces a minimum trade size to block micro-amount rounding exploits. The practical implication is that many “weird” tokens are either handled via balance-delta accounting or rejected atomically.
For many token types the Vault correctly handles accounting, but the current Routers do not support them and will revert when using them, because the amounts will not match.
Double entry point tokens are not compatible with the vault, because the transient accounting model would allow token deposits to be double counted. These should never be used.

7. Flash Loan and Economic Attacks

Pools could be targeted with large flash loans or economic exploits that manipulate balances or yield calculations.
Risks:

1. Temporary large liquidity swings causing mispricing or losses for LPs.
2. Exploitation of boosted pools or wrapped tokens through rapid entry/exit.
3. Multi-protocol attacks combining flash loans and external DeFi mechanisms.

We have analyzed adversarial scenarios (including flash loan style) to check invariants, limits, session fee management, price bounds and that accounting remains consistent under extreme, short lived liquidity changes. The Vault's unlock model enforces that all transient token deltas must be settled by the end of the interaction, so a flash-loan-funded sequence cannot "walk away" with unpaid debt. If anything remains unsettled, the call reverts. Swap execution also enforces user-supplied limits and minimum trade sizing, reducing profitability of rounding-based micro-manipulations and ensuring large price moves cannot silently execute beyond specified bounds. The code also explicitly anticipates flash-loan "round-trip" patterns by introducing a session/interaction concept to apply round-trip fees and guardrails within a single unlock interaction without accidentally taxing unrelated bundled actions. This is meant to make such attacks harder and more expensive while preserving aggregator composability. Overall, the economic safety story is "atomic settlement + enforced limits + anti-rounding guardrails."

8. Slippage and High-Volume Trade Risk

Large trades relative to pool liquidity can result in significant slippage, reducing execution efficiency.
Risks:

1. Traders executing large orders may face unexpected losses.
2. Pool token ratios may be temporarily skewed.
3. Arbitrageurs could exploit temporary imbalances.

We have reviewed slippage and limit checks in swap and routing logic, and tested extreme trade sizes and boundary conditions to ensure user-provided protections are enforced and that execution remains consistent under high-volume conditions. User slippage protection is enforced as a hard condition : routers pass `minAmountOut` / `maxAmountIn` as limitRaw, and the vault reverts whenever the realized raw amounts violate those bounds. To reduce stale-execution risk, routers also include a user-specified `deadline` and revert if the block timestamp exceeds it. The minimum swap size enforced by vault is in scaled units to prevent tiny trades that could otherwise amplify rounding edge cases, which is relevant for high-frequency or adversarial execution patterns. In practice, high-volume trade risk is primarily managed by strict limits and deadlines (user/ pools and integrator responsibility) plus deterministic on-chain enforcement that turns "bad slippage" into a revert rather than a silent loss.

9. Governance and Admin Privileges

Certain parameters in pools and vaults may be adjustable via governance or admin keys, including fees, pool parameters, and pausing.
Risks:

1. Malicious governance proposals could alter vault or pool behaviour.
2. Admin keys could be compromised or misused.

The code concentrates privileged entrypoints in `VaultAdmin`/`VaultExtension` and protects them with both structural and authorization gates : they can only be executed via the main Vault's `delegatecall` path (`onlyVaultDelegateCall`), and sensitive actions additionally require authenticate/role checks that resolve to Authorizer `actionId` permissions scoped to a specific pool/target (`_authorizer.canPerform(...)` in VaultAdmin). Emergency controls are also constrained in time and scope (pause window / buffer period logic, plus pool-specific role accounts such as `pauseManager` and `swapFeeManager`), which reduces the blast radius of a compromised key and makes privileged state changes predictable and auditable.

Vault calculations and safeguards

The Vault is the single accounting and settlement layer for all pools. Its core safety goal is to ensure that all per-token accounting interactions are fully settled, otherwise the interaction reverts. The Vault also tolerates unexpected token surpluses by absorbing them into reserves to prevent DoS via balance/reserve desynchronization.

Canonical balances and scaling

The Vault normalizes token amounts into a single internal 18-decimal fixed-point format by applying token-decimal scaling and (when configured) rate conversions. This ensures all pool math is performed in a consistent unit system.Whenever a conversion requires rounding, the Vault uses conservative rounding directions chosen to avoid overstating pool balances or user entitlements, reducing the chance that an attacker can profit from repeated rounding at boundaries. Minimum-amount checks further prevent dust-sized operations from turning rounding differences into extractable value.For swaps specifically, the Vault computes the pool's current live scaled balances by applying rates and rounding down when loading/reloading `balancesLiveScaled18` (`Rounding.ROUND_DOWN`), so the Vault never treats a token balance as larger than what can be safely accounted for.

Transient accounting and final settlement

All Vault actions execute inside a single `unlock()` "session". During the session, the Vault does not immediately require each intermediate step to be balanced, instead it records, for each token, a net delta (credit or debt) representing how much the session owes to the Vault or is owed by the Vault.When `unlock()` completes, the Vault requires that every token's net delta is fully settled (i.e., the final net delta is zero). If any token remains unsettled, meaning the session would leave the Vault with an unaccounted deficit or surplus, the entire interaction reverts with `BalanceNotSettled()`. This is the Vault's core accounting safeguard: complex multi-step swaps/joins/exits (including external calls via hooks) must still end in a globally balanced, settled state.

Additionally BPT are non-transient in V3 batch swaps, preventing the `exitSwap` vector that allowed callers to drain any V2 pool to very low liquidity levels by 'borrowing' BPT they didn't hold, and settling at the end. In V3 batch swaps, BPT are burned immediately, mid-transaction, instead of at the end.

Swap/join/exit amount computation boundaries
The Vault validates operation structure (e.g., pool registration/initialization, token index resolution, and input array length matching) and enforces user-provided limits (e.g., swap `limitRaw`, `maxAmountsIn`, `minAmountsOut`, `minBptAmountOut`, `maxBptAmountIn`) so pool math cannot bypass caller slippage constraints.
As an additional safeguard, the Vault enforces a minimum amount threshold on swaps and on non-zero liquidity amounts (BPT and per-token scaled amounts), while still allowing zero amounts where required to support certain single-token liquidity patterns.

Fee accounting and isolation
Swap fee logic computes fees during swaps. The aggregate fee portion (protocol + pool creator) is tracked separately in `_aggregateFeeAmounts` and deducted from the pool's token-in balance, while the remaining LP fee stays in the pool. Fees are applied via explicit accounting updates to Vault/pool balances rather than implicit reliance on token balance changes.

Reentrancy and call-graph control
The Vault uses `nonReentrant` guards on critical paths and enforces execution within an unlocked "session" so that any external-call effects must still satisfy the end-of-unlock settlement invariant.

Hook and pool containment
Hooks can execute external logic during swaps/liquidity flows, but the Vault relies on (a) nonReentrancy on critical state-updating paths and (b) the must-settle transient accounting requirement. After a hook call that can potentially change data, the Vault correctly reloads all relevant data.

ERC4626 buffer and wrap/unwrap correctness
When interacting with ERC4626 wrapped tokens, the Vault updates reserves after wrap/unwrap operations by checking post-call balances against expected reserves. Surpluses are absorbed into reserves (anti-DoS), while deficits revert (NotEnoughUnderlying/NotEnoughWrapped), enforcing ERC4626 compliance assumptions at the Vault boundary. One thing to mention for ERC4626 buffers is the fact that they are explicitly initialized once (`initializeBuffer`, locking

the wrapper's underlying asset) and the Vault gates buffer entry points on initialized state and rejects re-initialization.

Input validation and user limits
Swaps enforce nonzero amounts, forbid `tokenIn == tokenOut`, apply a minimum trade amount guard, and enforce user-provided bounds via `SwapLimit(...)`. Liquidity operations similarly check min/max bounds (e.g., `AmountInAboveMax`, `AmountOutBelowMin`, `BPT min/max`).

Pool calculations and safeguards

In V3, pools have an explicit one-time initialization step (`initialize`) after registration that seeds initial balances and mints initial BPT. Vault then marks the pool as initialized and rejects any subsequent initialization attempts, structurally preventing re-initialization (re-seeding) attacks.

Stable Pool
Stable Pools are designed for assets that trade near parity (e.g., stablecoins), using the Stableswap invariant to minimize slippage. The math Invariant satisfies $An^nS + D = ADn^n + \frac{D^{n+1}}{n^n \prod_{i=0}^{n-1} x_i}$ and is calculated using the Newton-Raphson approximation for the invariant D where *A* is the amplification parameter, *n* is the number of tokens, and x_i are token balances.

Swaps : `computeBalance(A, balances, D, tokenIndex)` solve balance y , the post-swap balance of one token given the invariant D and all other balances (using Newton-Raphson on a quadratic rearrangement) : $y \leftarrow \frac{y^2 + c}{2y+b - D}$ and where ±1 are deliberate rounding biases (protocol-favorable) :

- Exact In : $\Delta y = b_j - y - 1$ for a swap from $token_i$ to $token_j$ where $b_i = b_i + \Delta x$
- Exact out : $\Delta x = x - bi + 1$ where $b_j = b_j - \Delta y$

Key Parameters/value Safeguards :

- Amplification Parameter A has a supported range from 1 to 50.000 to specify the curvature/flatness of the stableSwap invariant.

- Amp update rate-limits: duration more than or equal to 1 day with max rate change less than or equal to 2× per day (per update schedule).
- Invariant Ratio: The invariant can grow or shrink within [60%,500%] during non-proportional operations.
- Max Tokens: 5 tokens per pool.
- Rounding directions always favor the protocol.
- Convergence: Computations revert if the iterative invariant or balance calculations do not converge within 255 steps.
- Swap fee bounds (stored with 24-bit precision in the Vault): 0.0001% min, 10% max.

A detailed analysis of the stable pool calculations can be found in Appendix B

Weighted pool

Weighted Pools implements the Constant Weighted Product formula, allowing for arbitrary asset weights (e.g., 80/20 pools). The math Invariant is $V = \prod_{i=0}^{n-1} B_i^{w_i}$ $with \sum_{i=0}^{n-1} w_i = 1$ where V is the invariant, B_i are balances and W_i are normalized weights summing to 1.

Swaps : `computeBalanceOutGivenInvariant` solves for a token balance given an invariant ratio: $newBalance = currentBalance \cdot invariantRatio^{\frac{1}{weight}}$ where $invariantRatio = \frac{invariant_{new}}{invariant_{old}}$

- Exact In : $amountOut = balanceOut \cdot (1 - (\frac{balanceIn}{balanceIn + amountIn})^{\frac{weightIn}{weightOut}})$
- Exact Out : $amountIn = balanceIn \cdot ((\frac{balanceOut}{balanceOut - amountOut})^{\frac{weightOut}{weightIn}} - 1)$

Key Parameters/value Safeguards :

- Min normalized weight is 1% (imposing a max 100:1 weight ratio).
- Swap fee bounds (stored with 24-bit precision in the Vault): 0.001% min, 10% max.
- Invariant ratio bounds for non-proportional liquidity ops: [70%, 300%], chosen to keep power/exponentiation within safe numerical bounds.
- Swap amount bounds: max in/out is 30% of the current balance (reverts on MaxInRatio/MaxOutRatio).
- Rounding in weighted math is chosen to favor the protocol and avoid profitable round trips.

- Minimum token balances are enforced (at least 1e6 scaled18, and higher for low-decimal tokens).
- Weighted BPT cannot be used as an internal rate provider (getRate() reverts), so it must not be nested as a WITH_RATE token.

A detailed analysis of the weighted pool calculations can be found in Appendix A

Gyro pool

Concentrated liquidity pools use circle (2-CLP) or ellipse (E-CLP) math to bound liquidity within specific price ranges.

1. **2CLP Pools**

Gyro 2 Tokens Concentrated liquidity pools (2CLP) use a price range [α,β] encoded as $\sqrt{\alpha}$, $\sqrt{\beta}$ with $\sqrt{\alpha} < \sqrt{\beta}$.

Math Invariant start from this simple equation $L^2 = (x + a)(y + b)$ where L is the pool invariant, (x,y) the real token balances and (a,b) the virtual offsets derived from the invariant and price bounds.

Swaps : 2-CLP actually computes L by solving a quadratic formula where

$$a = \frac{L}{\sqrt{\beta}} \text{ and } b = L\sqrt{\alpha}$$

- Exact In : $amount_{out} = \frac{(y+b)\circ\Delta x}{(x+a)+\Delta x}$
- Exact Out : $amount_{in} = \frac{(x+a)\circ\Delta y}{(y+b)-\Delta y}$

Key Parameters/value Safeguards :

- Swap fee bounds (stored with 24-bit precision in the Vault): 0.0001% min, 100% max (slightly below).
- Virtual-offset rounding is chosen conservatively (increases required input / decreases output).
    - Quadratic Bhaskara solution derivation.

2. **ECLP Pools**

Gyro Elliptic concentrated liquidity pools (ECLP) are also restricted to exactly 2 tokens and configured at deployment by five primary parameters: price bounds α and β defining the admissible trading range, a rotation angle φ (stored as c = cos(-φ) and s = sin(-φ) with $s^2 + c^2 = 1$), and a stretch factor λ ≥ 1 , more derived parameters (τα, τβ, u, v, w, z, dSq) are also computed off-chain at 38-decimal precision and used for calculations.

The math Invariant r takes the form: $g(t) = (c \circ A \circ v)(x, y) = r^2$ where $c(x, y) = x^2 + y^2$ is the circle equation , $v(x, y) = (x - a, y - b)$ that shifts reserves by virtual offsets, the matrix $A = Str(1/\lambda) \circ Rot(-\phi)$ is the linear transformation combining stretch and rotation and $t = (x, y)$ represents real token reserves.

Swaps : swaps are computed by solving a quadratic in transformed coordinates (from ellipse to circle via rotation + stretch) , it intentionally overestimates the post-swap reserve to be conservative :

- Exact In : $y' = \frac{-s\cdot c\cdot\bar{\lambda}\cdot x' - \sqrt{D}}{1-\bar{\lambda}\cdot s^2}$ where $\bar{\lambda} = 1 - 1/\lambda^2$ and $x' = x - a$
- Exact Out : $x' = \frac{-s\cdot c\cdot\bar{\lambda}\cdot y' - \sqrt{D}}{1-\bar{\lambda}\cdot c^2}$, the implementation computes an overestimate of the output balance through sign-aware rounding at each step

Key Parameters/value Safeguards :

- Overestimation strategy of error ensures protocol favoring outcomes for all liquidity operations.
- Invariant returned with certified error bounds: `computeInvariant(roundDown)=r-err, roundUp=r+err`
- Strong anti-overflow limits are enforced : Max balances [1e34] and Max Invariant [3e37]
- Unbalanced liquidity invariant-ratio bounds : $0.60 \leq \frac{r_{new}}{r_{old}} \leq 5.00$.
- Swap fee bounds (stored with 24-bit precision in the Vault): 0.00001% min, 100% max(slightly below).
- Rotation vector normalization: $|s2 + c2 - 1| \leq 1e^{-15}$
- Derived parameters normalized with accuracy $1e^{-15}$ in 38 decimal precision.

A detailed analysis of the Gyro pool calculations can be found in Appendix C.

Other pool types

Cow Pools are identical to weighted pool (same invariant and swap equations) but enforce routing/permissions via hooks :

- Swaps only allowed from a trusted CoW router
- Donations are enforced via a modifier
- Unbalanced liquidity is disabled at registration time

Surge pools do not change base AMM curve but change the effective swap fee via hooks and can also block unbalanced liquidity operations when surging..

LBP (Liquidity Bootstrapping pool) are weighted pools with time varying weights , swaps then use the same weighted swap equations.

Seedless LBP adds a constant virtual reserve balance v into the reserve balance for invariant/swaps but forbids paying out more reserve than the real on-chain reserve balance.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	0	0	0
Medium	1	1	1
Low	1	0	0
Informational	2	2	2
Total	0	0	0

Severity Matrix

Impact	Critical	Medium	Medium/High	High	Critical
	High	Low/Medium	Medium	Medium/High	High
	Medium	Low	Low/Medium	Medium	Medium/High
	Low	Informational	Low	Low/Medium	Medium
		Rare	Unlikely	Likely	Very Likely
		Likelihood			

Detailed Findings

ID	Title	Severity	Status
M-01	Incorrect fixed-point threshold	Medium	Fixed
L-01	Incorrect rounding direction for pre swap balances	Low	Pending
I-01	Missing lower-bound validation for E-CLP stretch factor	Informational	Fixed
I-02	Unchecked narrowing casts	Informational	Fixed

Medium Severity Issues

M-01 Incorrect fixed-point threshold		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Unlikely**
Files: weightedMath.sol	Status: Fixed	

Description: In `computeBalanceOutGivenInvariant`, comparing invariantRatio against the literal 1 (instead of the fixed-point unity `FixedPoint.ONE`) can systematically select the wrong division rounding mode for the 1/w exponent when invariantRatio is in 18-decimals fixed-point, causing biased `powUp` inputs and therefore incorrect balanceRatio and newBalance outputs, which materially distort liquidity add/remove accounting.

Note From Certora : This issue is only relevant for Weighted pools whose normalized weights are not exactly representable in the protocol's 18-decimal fixed-point domain (e.g., weights derived from fractions that cannot be represented precisely with 18 decimals). For "clean" weights (e.g., 50/50, 80/20, etc.) and other types of pools, the practical impact is further reduced because exponent inputs and rounding boundaries are more stable.

Recommendations: Replace the threshold invariantRatio > 1 with invariantRatio > FixedPoint.ONE to ensure the monotonicity-based rounding direction is chosen according to the correct fixed-point representation of unity.

Customer's response: Fixed: The Balancer Team notified Certora about this before the beginning of the audit after an extensive mathematical analysis and testing to prove that this error rounding could not cause the "overall" rounding to be wrong. Essentially, though rounding the exponent in the wrong direction did affect the final result somewhat, the `MAX_ERROR` correction in the `pow` function always overwhelmed any effects from the exponent rounding.

Fix Review: The patch is correct because it restores a domain-consistent unity comparison, ensuring the rounding-direction logic.

Low Severity Issues

L-01 Incorrect rounding direction for pre swap balances		
Severity: **Low**	Impact: **Low**	Likelihood: L**ikely**
Files: Vault.sol	Status:	

Description:
During swap operations, the pre-swap balances for both input and output tokens are calculated using the same rounding direction (rounding down). In a strictly optimal implementation, rounding directions should always favor the protocol to prevent any potential value extraction. The current implementation applies downward rounding uniformly to both tokens, which creates a small rounding discrepancy that could be interpreted as slightly favoring users over the protocol. The rounding error introduced by this approach is minimal—bounded to a maximum of 1 wei on the scaled 18-decimal representation of the balance.

Through comprehensive analysis of the protocol's complete fee structure, mathematical guardrails, and pool implementations, it has been demonstrated that this rounding behavior cannot be exploited to extract value from the protocol. The combination of:

- Swap fee mechanisms
- Yield fee calculations
- Protocol-wide guardrails
- Pool-specific invariant protections

Creates multiple layers of defense that ensure the theoretical rounding discrepancy remains unexploitable and economically irrelevant.
Moreover the added rounding-focused tests (`WeightedMathRoundingTest` and `BalanceAdjustmentRoundingTest`) explicitly compare the current "both balances rounded down" behavior against a protocol-favoring alternative (e.g., `balanceIn + 1` / `balanceOut - 1`) and show the resulting difference is at most 1 wei while always shifting outcomes in the protocol's favor. This confirms the rounding-direction choice here is a bounded dust effect with no realistic profit path, and any residual discrepancy is dominated by fees, limits, and invariant checks applied later in the pool calculations.

Recommendations:
Since the Vault is not upgradeable and the analysis confirms this rounding behavior is not economically relevant for an attacker, we recommend documenting it as a known design characteristic and choice for transparency.

Customer's response: For Weighted Pools, this is corrected at the pool level by adding +1 to balanceIn before swap calculations. For Stable and Gyro pools, analysis shows that uniform rounding up of tokenIn would not consistently favor the vault, and the existing fee mechanisms and guardrails provide sufficient protection.

Informational Severity Issues

I-01. Missing lower-bound validation for E-CLP stretch factor

Description: `GyroECLPPool/GyroECLPMath` accept `params.lambda` without enforcing Lambda >= 1 (in the protocol's fixed-point domain), while multiple computations and comments implicitly assume it for correctness and conservative rounding. Permitting lambda < 1 can lead to unexpected reverts or out-of-spec pricing/invariant behavior for misconfigured pools.
Recommendation: Enforce a strict lower bound at pool creation (e.g., `require(params.lambda >= 1e18)`).

Balancer's response: [Fixed in #PR1616.]

Fix review: Fixed.

I-02. Unchecked narrowing casts

Description: The code performs unchecked narrowing casts from `uint256 -> uint32` when storing CowSwap order deadlines and from uint256 -> int256 when converting the Router's quoted bptAmountOut inside PriceImpactHelper. While these values are expected to be within range in practice (timestamps < 2106, quotes far below int256.max), the absence of explicit bounds checks means extreme inputs/returns could silently truncate/overflow and lead to incorrect behavior.
Recommendation: Replace raw casts with checked conversions.

Balancer's response: [Fixed in #PR1616.]()

Fix review: Fixed.

Test Suite Review

Review Goals

The purpose of this review is to assess the current test coverage for Balancer V3 and evaluate the overall effectiveness of the test suite. In addition, this review identifies opportunities for improving both coverage and test quality to ensure more robust validation of system behavior.

Commit: 7ad0e280dda8624c6b593aa92099ebad7c6a5e3e

Contracts and Coverage

Vault.sol, VaultAdmin.sol, VaultCommon.sol, VaultStorage.sol

- **Coverage: VaultCore total coverage 99.2%**

File	Functions	Lines	Branches
Vault.sol	31/31 (100%)	353/362(97.5%)	72/79(91.1%)
VaultAdmin.sol	61/61(100%)	263/269(97.8%)	36/42(85.7%)
VaultCommon.sol	33/34(97.1%)	103/105(98.1%)	16/16(100%)
VaultStorage.sol	6/6(100%)	12/12(100%)	0/0(N/A)

- **Missing tests, which should be added:**
  - To reach 100% line coverage : constructor misconfiguration reverts, some reverts scenario on swap input, validation and simply input/validation reverts.
  - Good To Add : Add adversarial hook + accounting settlement invariants to validate safety under complex scenarios within transaction manipulations.
- **Expected coverage after implementing the suggested changes:** 100%.
- **Fix Review:** Pending

WeightedPool.sol and WeightedMath.sol

- **Coverage:**
  - WeightedPool.sol functions coverage is 94.4%, 124/146 lines are tested.
  - WeightedMath.sol functions coverage is 100%, 28/32 lines are tested.

- **Missing tests, which should be added:**
  - To reach 100% coverage : some constructors revert scenarios tests.
  - Good To Add : Round-trip no-profit scenarios under extreme weights and balance scales.
- **Expected coverage after implementing the suggested changes:** 100%.
- **Fix Review:** Pending

StablePool.sol and StableMath

- **Coverage:**
  - StablePool.sol functions coverage is 100%, 98/108 lines are tested.
  - StableMath.sol functions coverage is 100%, 51/55 lines are tested.
- **Missing tests, which should be added:**
  - To reach 100% lines coverage : some specific revert scenario paths regarding Amp.
  - Good To Add : Round-trip no-profit fuzzing to harden amp parameter transitions and numerical edge cases.
- **Expected coverage after implementing the suggested changes:** 100%.
- **Fix Review:** Pending

Gyro2CLPPool.sol

- **Coverage:**
  - Gyro2CLPPool functions coverage is 100%, 63/64 lines are tested.
  - Gyro2CLPMath functions coverage is 87.5%, 39/43 lines are tested.
- **Missing tests, which should be added:**
  - To reach 100% coverage : calcSpotPriceAinB reachability, some unhit revert branches in the maths.
  - Good To Add : Extend with ExtremeAmounts + imbalance/rate-provider rounding scenarios to stress 2-CLP's numerical safety at curve and scaling boundaries.
- **Expected coverage after implementing the suggested changes:** 100%.
- **Fix Review:** Pending

GyroECLPPool.sol, GyroECLPMath.sol

- **Coverage:**
  - GyroECLPPool functions coverage is 91,7%, 74/76 lines are tested.

- GyroECLPMath functions coverage is 71%, 195/244 lines are tested.
- **Missing tests, which should be added:**
  - To reach 100% coverage : A particular computeBalance call seems to not be reached.
  - Good To Add : extend with ExtremeAmounts + imbalance/rate-provider rounding scenarios to harden edge-of-curve correctness.
- **Expected coverage after implementing the suggested changes:** 100%.
- **Fix Review:** Pending

Disclaimer

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.

Appendix A: Analysis of Weighted Pool calculations

Introduction

Balancer V3 Weighted Pools implement the Constant Weighted Product (CWP) formula, a generalization of the Uniswap $x * y = k$ invariant that supports

- Multiple tokens (up to 8 tokens)
- Arbitrary weights (with $\sum_{i=0}^{n-1} w_i = 1e18$)
- Fixed weights that cannot change after deployment

In V3, all math is done over **18-decimal fixed point** values, and balances are passed in already **scaled-to-18** (and rate-adjusted) by the Vault (`balancesLiveScaled18`: post yield-fees, decimal scaling, and token rates).

The core invariant is: $V = \prod_{i=0}^{n-1} B_i^{w_i} \; with \; \sum_{i=0}^{n-1} w_i = 1$

Where:

- V = invariant (value function) is the pool's "value function" in the model; for weighted pools, it behaves like a geometric mean of balances where weights define the portfolio composition (e.g., 80/20).
- $balance\_i$ = token balance (scaled to 18 decimals)
- $weight\_i$ = normalized weight

This invariant ensures:

1. **Swap invariance (ignoring fees)**: swaps should not decrease V; with fees, V can increase.

2. **Linearity**: Proportional balance increases yield proportional invariant increases:
   $inv(a \; * n, b * n) \; = \; inv(a, b * n)$

   This property is essential for **fungible LP shares**: proportional deposits should mint proportional BPT.

3. **Custom Exposure**: LPs can maintain a portfolio with specific exposure (e.g., 80/20 WBTC/WETH) rather than forced 50/50.

4. **Impermanent Loss Control**: Higher weights on one asset result in lower Divergence Loss (IL) for that asset relative to a 50/50 pool.

5. **Zero-impact prices** are determined by the ratio of balances and weights: $P_j^i = \frac{(B_j * W_i)}{(B_i * W_j)}$

Core Parts of WeightedPool.sol and WeightedMath.sol

Fixed-Point Arithmetic Foundation

- `FixedPoint.ONE = 1e18` is the "1.0" unit. Multiplications/divisions rescale by `ONE`.
- Balances are assumed to already be in "`scaled18`" space : token decimals/rates are normalized by Vault helpers (conceptually as in `ScalingHelpers`).

Power Functions

Computation : $x\^y$ via $LogExpMath.pow(x, y) = exp(y * ln(x))$, then apply and explicitly one-sided relative error envelope :

- `MAX_POW_RELATIVE_ERROR` $= 1e^{-14}$ stored as 10000 in 1e18 terms which will results in a real error to be < $1e^{-14}$
  With the imposed limit of y×ln(x)=130, the computed maximum error has been verified to be ~ $1.12e^{-14}$, slightly more then the `MAX_POW_RELATIVE_ERROR`
  For weighted pools, the maximum y value is 99, which lowers the maximum achievable error to ~$1.5e^{-15}$, well below the $1e^{-14}$ limit
- powUp returns **raw + maxError** where maxError is derived from `MAX_POW_RELATIVE_ERROR` (guaranteed ≥ true value)
- powDown returns **raw - maxError** where maxError is derived from `MAX_POW_RELATIVE_ERROR` (floored at 0; guaranteed ≤ true value)

Power boundaries

- `MIN_NATURAL_EXPONENT = -41e18` and `MAX_NATURAL_EXPONENT = 130e18` constrain the domain of the exponentiation to avoid overflows or underflows in the internal 20-36 decimal fixed-point logic.
- Effective balance range: $10^{-18} < balance < 2^{188.56}$
- Maximum balance: $2^{128} - 1$(unconstrained by power bounds)

This is the foundation for protocol-favoring rounding: higher-level formulas pick up/down directions consistent with monotonicity so users never benefit from approximation/rounding.

Safety and Ratio Limits

Swap Bounds (30% limits) prevents bases from approaching power function extremes

- `MAX_IN_RATIO = 30e16` (30%) and `MAX_OUT_RATIO = 30e16` (30%) :

Invariant Ratio Bounds constrains $0.7 < invariantRatio < 3.0$ range for safe $invariantRatio^{\frac{1}{weight}}$ computation

- `MAX_INVARIANT_RATIO` $= 300e^{16}$ (300% growth)
- `MIN_INVARIANT_RATIO` $= 70e^{16}$(70% shrink)

Weight Constraints:

- `MIN_WEIGHT = 1e16` (1%)
- Maximum weight ratio: 100:1 (due to 1/weight ≤ 100)

Computing the Invariant

The pool calculates the invariant using the `computeInvariant` function which delegates to WeightedMath. Depending on the context (e.g., measuring pool growth vs. ensuring solvency), it can round up or down : $V = 1e^{18} \cdot \prod_i (B_i)^{W_i}$

Rounding Direction

- **computeInvariantDown**: Uses `powDown` and `mulDown` which underestimates V when calculating the available invariant given current balances. Used when lower invariant favors protocol.
- **computeInvariantUp**: Uses `powUp` and `mulUp` which overestimates V to requires slightly more assets from the user to mint a given amount of BPT.

Why this is safe

- **Adding liquidity**: Protocol computes `currentInvariant` UP and `newInvariant` DOWN which minimize invariantRatio and results in fewer BPT minted.

- **Removing liquidity**: Protocol computes invariantRatio UP which results in user burning more BPT.

Also note an important design choice: `WeightedPool.getRate()` is disabled. The contract explicitly warns that the invariant-based rate has **non-trivial, non-linear error** and without a caller-specified rounding direction it could become manipulable (so nesting as a "WITH_RATE" token is disallowed).

Both paths revert if `invariant == 0` (indicates zero balance).

Computing the balances

computeBalanceOutGivenInvariant solves for a token balance given an invariant ratio:

$newBalance = currentBalance \cdot invariantRatio^{\frac{1}{weight}}$ where $invariantRatio = \frac{invariant_{new}}{invariant_{old}}$

Rounding Strategy:

- Rounds UP overall (both exponent and multiplication) to return a balance that is conservatively high
- For invariantRatio > FixedPoint.ONE : `divUp(FixedPoint.ONE, weight)` (rounds exponent UP to increases monotonically)
- For invariantRatio < FixedPoint.ONE : `divDown(FixedPoint.ONE, weight)` (rounds exponent DOWN to decrease monotonically)

Swap Functions

Exact In (computeOutGivenExactIn)

$$amountOut = balanceOut \cdot (1 - (\frac{balanceIn}{balanceIn + amountIn})^{\frac{weightIn}{weightOut}} - 1)$$

Rounding choices are deliberately mixed based on monotonicity to ensure `amountOut` is **rounded down overall** :

- cap: $aIn <= 0.3 * bIn$
- base: $bIn/(bIn + aIn)$ is computed with `divUp` (slightly larger base)
- exponent $wIn/wOut$ computed with `divDown` (slightly smaller exponent)
- power computed with `powUp` (slightly larger power since base < 1)
- complement used to avoid underflow if rounding pushes power above 1
- multiply by `bOut` with `mulDown`

All of these choices converge to: user gets **<=** the ideal real-number output.

Exact Out (computeInGivenExactOut)

$$amountIn = balanceIn \cdot ((\frac{balanceOut}{balanceOut - amountOut})^{\frac{weightOut}{weightIn}} - 1)$$

Rounding choices ensure amountIn is **rounded up overall** :

- cap: $aOut <= 0.3 * bOut$
- base: $bOut/(bOut - aOut)$ uses `divUp` (larger base)
- exponent uses `divUp` (larger exponent)
- power uses `powUp` (larger power)
- subtract 1, then multiply with `mulUp`

Net effect: user pays **>=** the ideal real-number input.

Handling Precision and rounding safety

Conservative Overestimation

Every rounding choice favors the protocol. For any "round trip" (add then remove, or swap A→B→A), the user receives ≤ original tokens.

Bounded Power Function

The hardest part of weighted math is x^y. V3 explicitly acknowledges approximation error and forces it to be one-sided with `MAX_POW_RELATIVE_ERROR` +1 wei padding to ensures strict one-sidedness even at very small values, so callers can treat `powUp` as a guaranteed upper bound and `powDown` as a guaranteed lower bound.

This is the core mitigation against “round-trip profit via numerical error”: if every step is biased against the trader (or in favor of the protocol), repeated paths can’t accumulate dust into profit.

Weight Limits : Minimum 1% weights ensure:

- Maximum $100:1$ weight ratios
- Bound $1/w$, preventing exponent explosions.

Invariant Ratio Bounds : The Vault's BasePoolMath enforces $0.7 - 3.0$ ranges

- `ensureInvariantRatioBelowMaximumBound`: Add operations cannot increase invariant by $> 300\%$

- `ensureInvariantRatioAboveMinimumBound`: Remove operations cannot decrease invariant by $> 30\%$

These prevent `computeBalance` from hitting `LogExpMath` domain errors.

Swap Ratio Limits : Single swaps are capped at 30% of pool balance

- Prevents extreme price impact exploitation
- Bases stay within safe ranges for $ln()$ and $exp()$ operations
- Keeps power function inputs well within valid domain
- Forces large trades to split across multiple transactions

Overflow Protection

- Solidity 0.8.x provides built-in overflow checking
- LogExpMath uses higher precision intermediates (**20/36 decimals**) to minimize truncation
- Large exponent operations are bounded by the domain check on $y * ln(x)$
- `FixedPoint.mulUp/mulDivUp` use careful formulations to avoid intermediate overflow patterns (and explicitly revert on division-by-zero).
- `FixedPoint.complement()` clamps $1 - x$ at 0, preventing negative intermediate values when `powUp` overshoots above 1 (important in `ExactIn` where `base<1`).

Appendix B: Analysis of StablePool calculations

Balancer V3 stable pools implement a generalized StableSwap invariant that allows multiple tokens to maintain a tightly-coupled price curve near parity while allowing large liquidity. The invariant combines constant-sum behavior near equilibrium with constant-product behavior when pools are imbalanced.

For an n-token pool with amplification parameter

$$A$$

and total sum

$$S = \sum_{i=0}^{n-1} x_i$$

, the StableSwap invariant satisfies:

$$An^n S + D = ADn^n + \frac{D^{n+1}}{n^n \prod_{i=0}^{n-1} x_i}$$

This equation ensures:

1. Near equilibrium (balances similar), the pool behaves like a **constant sum**: low slippage.
2. When one token is far from equilibrium, the pool behaves like a **constant product**, preserving liquidity and avoiding negative balances.

Balancer V3 solves this invariant using fixed-point arithmetic on-chain, handling rounding and precision carefully.

Computing an Individual Token Balance (computeBalance)

Balancer V3 provides the `computeBalance` function to compute an unknown token balance

$$x_k$$

given the invariant

$$D$$

and all other balances:

The computation solves a quadratic-like equation derived from the invariant:

$$x_k^2 + bx_k + c = 0$$

where:

$$b = \sum_{i \neq k} x_i + \frac{D}{A'}$$

$$c = \frac{D^2}{A' \cdot P_D} \cdot x_k$$

$$A' = A \cdot n$$

(effective amplification)

$$P_D \approx \frac{n^n \prod_{i \neq k} x_i}{D^{n-1}}$$

(scaled product of other balances)

The function proceeds as follows:

1. Compute

   $$b$$

   and

   $$c$$

   from the invariant and other token balances.
2. Set an initial overestimated guess:

   $$x_k^{(0)} = \frac{D^2 + c}{D + b}$$

3. Iteratively refine using a Newton-like update:

   $$x_k^{(n+1)} = \frac{\left(x_k^{(n)}\right)^2 + c}{2x_k^{(n)} + b - D}$$

4. Stop when the change is ≤ 1 (integer fixed-point precision).

Math Freedom

- **Invariant * Invariant** : even on the biggest pool we have a 10e27 margin.
- `(tokenBalance * 2) + b - invariant` where if `invariant > (tokenBalance * 2) + b` would DOS but here it would require a really really big imbalance with a really low amp so invariant computation would fail before computeBalance.

Handling Precision and Rounding Safely

Conservative Overestimation

- The `c` term is intentionally overestimated by multiplying **after** rounding the division. This ensures the initial guess and iterations always start **above the true balance**, which guarantees monotonic convergence.

$$c = \lceil \frac{D^2}{A' \cdot P_D} \rceil \cdot x_k$$

- Overestimation prevents underflows or negative denominators in the iterative formula.

Monotonic Iteration

- The iterative update always approaches the solution from above:

$$x_0 > x_1 > x_2 > \ldots > x_{true}$$

- This avoids oscillations, ensures denominators remain positive, and prevents divergence.

Accumulating Products and Sums

- The partial product

$$P_D$$

  is computed iteratively with intermediate divisions. Though each division introduces truncation error, the overestimation in `c` compensates for this.
- Sum truncation is downward, which is safe because it slightly underestimates

$$b$$

  , making the Newton step slightly larger, again biasing the iteration conservatively.

Fixed-Point Arithmetic and Div-Up

- All divisions use `divUpRaw` to round **up**, ensuring that rounding errors never produce a lower-than-expected balance.
- This prevents invariant-breaking values even under extreme token imbalance.

Overflow Protection

- Multiplications in the numerator are scaled and rearranged to avoid exceeding 256-bit limits.
- By performing division before multiplying in sensitive terms, the algorithm reduces overflow risk while remaining conservative.
- With extreme imbalance and large swaps it is possible for an overflow to happen, reverting the swap. This will have no impact in normal operations.

Appendix C: Analysis of GyroPool calculations

2CLP Pool

Introduction

Balancer V3 **Gyro 2-CLP** (Two-asset Concentrated Liquidity Pool) implements an hyperbola concentrated liquidity AMM that restricts trading to a configurable price range [α,β] determined at deployment. Unlike traditional constant product AMMs such as Uniswap V2's $x \cdot y = k$ formula that distribute liquidity across the entire price range (0,∞), 2-CLPs achieve capital efficiency comparable to Uniswap V3 through a simpler mathematical construction based on **virtual reserves**. The pool holds exactly two tokens and is parameterized by the square roots of its price bounds, $\sqrt{\alpha}$ and $\sqrt{\beta}$ stored as immutables (`sqrtAlpha` and `sqrtBeta`) since most calculations naturally involve square roots, reducing gas costs.

In V3, all computations operate on **18-decimal fixed-point** values, and the Vault provides token balances already normalized into "live scaled 18" space (`balancesLiveScaled18`: post yield-fees, decimal scaling, and rate application). The pool's behavior is fully determined by the two live balances (x,y) and the fixed parameters $\sqrt{\alpha}$ and $\sqrt{\beta}$.

The core invariant is defined on **virtual balances**, which consist of the real token balances augmented by "virtual offsets":

$$L^2 = (x + a)(y + b)$$

Where:

- L is the pool's invariant representing total liquidity
- x and y are the real token balances (scaled to 18 decimals)
- a and b are the virtual offsets derived from the invariant and price bounds:

$$a = \frac{L}{\sqrt{\beta}} \text{ and } b = L \circ \sqrt{\alpha}$$

This invariant structure ensures the pool functions like a standard constant product market maker but with **effective reserves** of (x+a) and (y+b), resulting in tighter spreads and lower slippage for trades executed within the defined price interval. The spot price P, calculated as the ratio of virtual reserves $\frac{y+b}{x+a}$, is mathematically constrained to the range [α,β] : when x=0, the price reaches β, and when y=0, the price reaches α. Trading outside this range is impossible as one asset becomes fully depleted.

Operationally:

- **Invariant computation** (`calculateInvariant`) solves for L via a quadratic formula derived from rearranging the invariant equation with the [α,β] parameters.
- **Swaps** are executed using the virtual balances x′=x+a and y′=y+b, yielding constant-product-style closed forms for exact-in/exact-out without requiring L^2 directly (the product $x' \circ y'$ is used instead to prevent error accumulation).
- **Rounding direction** is chosen conservatively depending on context (e.g., invariant rounded up for `computeBalance` paths; virtual offsets rounded to favor the pool during swaps), ensuring users do not gain from approximation/rounding.

Like Weighted Pools, the 2-CLP invariant exhibits **linearity**: proportional changes to the real balances x and y result in proportional changes to the invariant L and the offsets a and b. This property is essential for the fungibility of LP shares, ensuring that proportional deposits mint proportional amounts of BPT.

For background on Gyro 2-CLPs and their intended price-range behavior, see the Gyroscope documentation: Gyro 2-CLPs.

Core Parts of 2CLP Pools

Fixed-Point Arithmetic Foundation

- `FixedPoint.ONE = 1e18` is the "1.0" unit. Multiplications and divisions are rescaled by `ONE` to maintain precision.
- Balances are assumed to already be in "scaled18" space: token decimals and rates are normalized by the Vault before being passed to pool math functions (`balancesLiveScaled18`).
- The library provides directional rounding variants (`mulDown`/`mulUp`, `divDown`/`divUp`) that enable protocol-favoring arithmetic throughout the computation pipeline.

Virtual Balances Architecture

The architectural concept that makes 2-CLP distinct is the use of **virtual balances**. Instead of trading directly against the real reserves (x,y), the pool trades against shifted reserves: $x' = x + a$, $y' = y + b$ where (a,b) are virtual offsets derived from the invariant.

This shift concentrates liquidity inside the intended pricing range [α,β]: within that range, the pool behaves as if it had deeper reserves (higher effective liquidity). As a result, swap execution uses constant-product-style closed forms on (x′,y′) while the pool's shape is governed by the range parameters.

Quadratic Math Architecture

Unlike Weighted Pools which rely on iterative power approximations via `pow()`, the 2-CLP invariant is solved analytically using the quadratic formula . The fundamental constraint $L^2 = (x + a)(y + b)$ expands into a standard quadratic form:

$$0 = (1 - \sqrt{\frac{\alpha}{\beta}})L^2 - (\frac{y}{\sqrt{\beta}} + x\sqrt{\alpha})L - xy$$

The coefficients satisfy $a > 0,\ b < 0,\ c \leq 0$, a special case that works cleanly with unsigned integers by representing −b as `mb` and −c as `mc`. The `Gyro2CLPMath` library implements:

- `calculateQuadraticTerms()`: Prepares coefficients with direction-aware rounding
- `calculateQuadratic()`: Applies Bhaskara's formula with `sqrt()` (Newton-Raphson, tolerance parameter 5)

Parameter Constraints

The constructor enforces a single critical constraint: $\sqrt{\alpha} < \sqrt{\beta}$. This ensures α<β , guaranteeing the lower price bound is strictly less than the upper. These parameters create hard mathematical boundaries: the spot price $P = \frac{y+b}{x+a}$ is constrained to [α,β]. At these limits, one asset is fully depleted, preventing further trading in that direction.

Safety and Ratio Limits

1. *Swap Fee Bounds:*
   - `minSwapFeePercentage = 1e12` (0.0001%)
   - `maxSwapFeePercentage = 1e18` (100%)

The minimum fee prevents liquidity approximation attacks where add/remove combinations could be more profitable than direct swaps.

2. *Invariant Ratio Bounds:* unlike Weighted Pools (0.7–3.0 range) or E-CLP (60%–500%), the 2-CLP does not impose strict invariant ratio bounds:
   - `getMinimumInvariantRatio()` returns `0`
   - `getMaximumInvariantRatio()` returns `type(uint256).max`

   This defers such constraints to the Vault layer or hook logic.

3. *Asset Bounds:* Swap functions enforce physical balance constraints, reverting with `AssetBoundsExceeded()` if `amountOut > balanceOut`. This defense-in-depth mechanism ensures virtual liquidity never promises more tokens than physically exist.

4. Rounding Directions : for the quadratic formula itself, the term $a = 1 - \frac{\sqrt{a}}{\sqrt{\beta}}$appears in the **denominator**, so rounding the invariant **up** requires rounding a **down** (by rounding the division $\frac{\sqrt{a}}{\sqrt{\beta}}$ **up**). This inverse relationship is explicitly handled in `calculateQuadraticTerms()`.

Math Freedom

The 2-CLP intentionally avoids expensive or highly non-linear primitives:

- **No exponentiation/logarithms**: Unlike Weighted Pools' `LogExpMath.pow()` with its $\pm 10^{-14}$ relative error envelope
- **Single square root**: Only during invariant computation via Newton-Raphson
- **Simple rational formulas**: Swap execution uses algebraically simple multiply/divide on virtual balances
- The b^2 term in the discriminant is computed in expanded form for better fixed-point precision: $b^2 = x^2\alpha + 2xy\sqrt{\frac{\alpha}{\beta}} + \frac{y^2}{\beta}$ rather than simply squaring $b = \frac{b}{\sqrt{\beta}} + x\sqrt{\alpha}$

Computing the invariant

The 2-CLP invariant L is defined implicitly through the equation $L^2 = (x + a)(y + b)$, where the virtual offsets $a = \frac{L}{\sqrt{\beta}}$ and $b = L \circ \sqrt{\alpha}$ and depend on L itself. This circular dependency requires solving a quadratic equation to obtain L from the real balances.

The Vault-facing entrypoint `computeInvariant` is a thin wrapper that fetches $\sqrt{\alpha}$, $\sqrt{\beta}$ and delegates to the math library.

Mathematical Derivation

Starting from the invariant definition and substituting the virtual offset formulas:

$$L^2 = (x + \frac{L}{\sqrt{\beta}})(y + L\sqrt{\alpha})$$

Expanding the right-hand side:

$$L^2 = xy + xL\sqrt{\alpha} + \frac{yL}{\sqrt{\beta}} + \frac{L^2\sqrt{\alpha}}{\sqrt{\beta}}$$

Rearranging into standard quadratic form : $AL^2 + BL + C = 0$:

$$(1 - \sqrt{\frac{\alpha}{\beta}})\, L^2 - (x\sqrt{\alpha} + \frac{y}{\sqrt{\beta}})\, L - xy = 0$$

This quadratic has a special structure that simplifies implementation. The coefficients satisfy:

$$A = 1 - \frac{\sqrt{\alpha}}{\sqrt{\beta}} > 0 \text{ (since } \alpha < \beta)$$

$$B = -(x\sqrt{\alpha} + \frac{y}{\sqrt{\beta}}) < 0 \text{ (since balances are positive)}$$

$$C = -xy \leq 0$$

Since A > 0, B < 0 and C ≤ 0, the implementation works with absolute values $mb = |B|$ and $mc = |C|$ to avoid signed arithmetic entirely. The positive root (the only physically meaningful solution) becomes:

$$L = mb + \frac{\sqrt{mb^2 + 4 \circ A \circ mc}}{2 \circ A}$$

The discriminant $mb^2 + 4 \circ A \circ mc$ is always non-negative for valid inputs since both terms are non-negative, guaranteeing a real solution exists.

Implementation Steps

The `calculateInvariant()` function delegates to two helper functions:

Step 1: `calculateQuadraticTerms()` computes the quadratic coefficients with direction-aware rounding.

Step 2: `calculateQuadratic()` applies Bhaskara's formula.

The only non-linear primitive required is a single square root of a non-negative radicand.

Rounding Direction : The implementation achieves directional rounding through function pointers that swap between `mulUp`/`divUp` and `mulDown`/`divDown`.

Approximations and Error Sources

All approximations come from two sources:

1. **Fixed-point rounding**: Each `mulDown`/`divUp` operation introduces error ≤1. The expanded computation contains 15 rounding operations (9 in b^2).
2. **Square root approximation**: `GyroPoolMath.sqrt()` uses Newton-Raphson iteration with tolerance parameter 5. The bi-directional function asserts the squared result is within a tight

band around the input, bounded by tolerance scaled by guess magnitude, over 1 million fuzzing test the invariant decreased by a max value of 50,001 wei 4 times in extreme narrow-range pools so if we only consider the minimum swap fee of 0.0001%, this is economically not exploitable.

Unlike E-CLP, the 2-CLP does not maintain explicit error bounds on the invariant. The simpler formula and absence of extra-precision (38-decimal) arithmetic mean error is bounded by a constant number of 1-wei rounding steps, negligible for practical balance magnitudes.

Math Freedom

The 2-CLP invariant computation has limited "math freedom" compared to iterative methods:

1. What is used:
    - Fixed-point `mul`/`div` with explicit rounding direction
    - Single Newton-Raphson square root (7 iterations, tolerance-checked)
    - Algebraic expansion for precision (b^2 terms computed separately)

2. Why This Is Safe
    - Structural constraints ensure well-defined computation: The pool enforces α<β at construction, guaranteeing $A = 1 - \frac{\sqrt{\alpha}}{\sqrt{\beta}} > 0$. This makes the denominator 2A well-defined and ensures the "no negatives" quadratic form is valid.
    - Non-negative discriminant guaranteed: Both terms are non-negative for valid inputs, so the square root is always real :

$$mb^2 + 4\,Amc = (x\sqrt{\alpha} + \frac{y}{\sqrt{\beta}})^2 + 4\,(1 - \sqrt{\frac{\alpha}{\beta}})xy$$

    - Monotonicity: The computed invariant is monotonic with respect to input balances. Increasing x or y strictly increases mb and mc, leading to a larger L. This prevents "rounding arbitrage" where users could manipulate balances to artificially inflate the invariant.
    - Reliance on the Invariant : The computed invariant L serves multiple critical purposes.
    - Virtual offset computation: The offsets used in swaps depend on L. An underestimated invariant produces smaller virtual offsets, affecting swap calculations. The rounding direction in `onSwap()` is `ROUND_DOWN`, ensuring the pool never overestimates its liquidity.
    - BPT minting/burning: During liquidity operations, the invariant ratio determines how many BPT tokens correspond to a given deposit or withdrawal. Protocol safety requires computing this ratio conservatively.

Computing An Individual Balance

The `computeBalance` function in `Gyro2CLPPool.sol` determines the required balance of one token given the other token's balance and a target invariant ratio. This function is invoked by the Vault during **non-proportional (unbalanced) liquidity operations**, where a user adds or removes liquidity in a single token rather than proportionally across both tokens.

When This Is Required

The Vault's `BasePoolMath` calls `pool.computeBalance(currentBalances, tokenIndex, invariantRatio)` in two scenarios, single token add (desired BPT to receive) and single token remove (BPT to burn).

In both cases, the protocol must overestimate the required deposit or underestimate the withdrawal to protect the pool from value extraction through rounding manipulation.

Mathematical Derivation

Starting from the 2-CLP invariant and given the current balances $x_{current}$ and $y_{current}$ a target `invariantRatio`, the new invariant is: $L_{new} = L_{new} \circ invariantRatio$

For **single-sided operations**, one balance remains unchanged while the other adjusts to satisfy the new invariant. The virtual offsets scale with the invariant : $a_{new} = \frac{L_{new}}{\sqrt{\beta}}$ and $b_{new} = L_{new} \circ \sqrt{\alpha}$

If token Y's balance remains at $y_{current}$ solving for x_{new} :

$$x_{new} + a_{new} = \frac{L_{new}^2}{y_{current} + b_{new}}$$

$$x_{new} = \frac{L_{new}^2}{y_{current} + b_{new}} - a_{new}$$

Symmetrically, if token X's balance is fixed:

$$y_{new} = \frac{L_{new}^2}{x_{current} + a_{new}} - b_{new}$$

This is a closed-form computation, no iteration, no root-finding beyond the invariant computation itself, so it is monotone and straightforward to bias via rounding.

Precision Detail: Raw Multiplication and Division

Two operations use "raw" arithmetic rather than `FixedPoint` methods:

> **`squareNewInv = invariant * invariant`**, this is intentionally a raw multiply. The invariant L is in 18-decimal fixed point, so L^2 is naturally 36-decimal. This matches the product $(x + a)(y + b)$ which is also 36-decimal (two 18-decimal terms multiplied).
>
> **`divUpRaw`** — The division is 36-decimal ÷ 18-decimal, which returns an 18-decimal result. The `divUpRaw` function performs ceiling division without rescaling.

Rounding Direction Analysis

The goal is to maximize **`newBalance`** so that:

- During additions: users deposit **more** tokens than theoretically necessary
- During removals: users receive **fewer** tokens than theoretically possible

Step	Operation	Rounding	Effect on **`newBalance`**
1	calculateInvariant	UP	↑ Larger L → larger L^2 in numerator
2	invariant.mulUp(ratio)	UP	↑ Larger L_{new} → larger L^2
3	Invariant * invariant	Raw	↑ Larger numerator
4	invariant.divDown(sqrtBeta)	DOWN	↑ Smaller a → subtract less
4b	invariant.mulDown(sqrtAlpha)	DOWN	↑ Smaller b → smaller denominator → larger quotient
5	divUpRaw($L^2, denom$)	UP	↑ Larger quotient before subtraction

The combined effect ensures newBalance is always an **overestimate** of the mathematically exact value.

Invariant Ratio Bounds : Unlike Weighted Pools (0.7–3.0 range) or E-CLP (60%–500%), the 2-CLP imposes no pool-level constraints on the invariant ratio.

Safety relies on:

- Vault-level supply arithmetic and fee logic
- Solidity 0.8 overflow/underflow reversion (if extreme invariantRatio causes invariant * invariant to overflow, the transaction reverts safely)
- The pool's own arithmetic bounds (e.g., AssetBoundsExceeded in swaps)

Why This Is Safe

- Algebraic exactness: The formulas are algebraically exact given the invariant definition. There is no iterative approximation—the only error sources are fixed-point rounding, which are bounded and directionally controlled.
- No division by near-zero: The denominator $(y + b)$ or $(x + a)$ is always positive
- Real balances $x, y \geq 0$ and virtual offsets $a, b > 0$
- Even if one real balance is zero, the virtual offset ensures a positive denominator
- Monotonicity: The computed balance is monotonic with respect to invariantRatio: $\frac{d(x_{new})}{d(ratio)} > 0$
- The rounding choices preserve this monotonicity, preventing erratic behavior for small input changes and eliminating "rounding arbitrage" opportunities.
- Consistency with invariant semantics: If the computed newBalance were substituted back into calculateInvariant along with the unchanged balance, the result would satisfy: $L_{recomputed} \geq L_{target}$
- This guarantees the pool's value is preserved or increased, never diminished by rounding exploitation.

Economic Safety

The Vault's BasePoolMath uses newBalance to compute user-facing quantities using $amountIn = newBalance - oldBalance$ for single token add and $amountOut = oldBalance - newBalance$ for single-token remove which ensures the dominant user-facing quantities move in the conservative direction: **more in on adds, less out on removes**—the same "monotone + one-sided rounding at the interfaces that matter" safety story used throughout the pool calculations.

ECLP Pool

Introduction

Balancer V3 Gyro E-CLP (Elliptic Concentrated Liquidity Pool) implements an AMM where trading occurs along a section of an ellipse curve. The construction is based on transforming a circle through stretching, rotation, and shifting operations, yielding a trading curve that can be precisely configured to concentrate liquidity within a specified price range. Unlike constant-product AMMs that trade along a hyperbola or StableSwap pools that interpolate between constant-sum and constant-product behavior, the E-CLP provides explicit geometric control over liquidity distribution through its parametrization.

The pool is restricted to exactly 2 tokens and is configured at deployment by five primary parameters: price bounds α and β defining the admissible trading range, a rotation angle φ (stored as c = cos(-φ) and s = sin(-φ) with $s^2 + c^2 = 1$), and a stretch factor λ ≥ 1.

When λ = 1 the trading curve is a circle section; as λ increases, the curve becomes progressively more elliptical, concentrating liquidity around the peg price tan(φ). The mechanism only permits reserve states on the admissible arc of the ellipse between the boundary points $(x^+, 0)$ and $(0, y^+)$, corresponding to prices β and α respectively.

The core invariant r takes the form: $g(t) = (c \circ A \circ v)(x, y) = r^2$

Where:

- $c(x, y) = x^2 + y^2$ is the circle equation
- $v(x, y) = (x - a, y - b)$ shifts reserves by virtual offsets
- $A = Str(1/\lambda) \circ Rot(-\phi)$ is the linear transformation combining stretch and rotation
- r is the liquidity invariant (radius of the underlying circle) ≥ 0
- $t = (x, y)$ represents real token reserves

The transformation matrix A and its inverse are:

$$A = \begin{bmatrix} c/\lambda & -s/\lambda \\ s & c \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} c\lambda & s \\ -s\lambda & c \end{bmatrix}$$

The E-CLP operates across three coordinate spaces.

1. Real reserves $t = (x, y)$represent actual token balances in the pool.

2. Shifted reserves $t' = (x - a, y - b)$ represent the position on an ellipse centered at the origin after subtracting virtual offsets.
3. Transformed reserves $t'' = A \circ t'$ represent the corresponding point on a circle centered at the origin.

The virtual offsets (a, b) position the ellipse such that the admissible trading arc intersects the positive quadrant with the correct price bounds:

$$a = r \circ (A^{-1} \circ \tau(\beta))_x$$

$$b = r \circ (A^{-1} \circ \tau(\alpha))_y$$

Here $\tau(p) = \eta(\zeta(p))$ maps a price p on the ellipse to a normalized point on the unit circle through two transformations:

1. ζ converts the ellipse price to a circle price via $\zeta(p) = \lambda \circ (-s + c \circ p) / (c + s \circ p)$
2. 2. η maps a circle price to a unit circle point via $\eta(p^c) = (p^c, 1)/\sqrt{(1 + (p^c)^2)}$.

A critical property is that **offsets scale linearly with r**, ensuring price bounds remain fixed when liquidity changes proportionally.

The dSq parameter captures the realized value of $c^2 + s^2$ in extra precision, correcting for any rounding error in the trigonometric components.

Validation enforces strict bounds: rotation vector normalization to $\pm 10^{-15}$, τ vectors normalized to $\pm 10^{-15}$ in extra precision, $\lambda \leq 10^8$, and anti-overflow constraints on the invariant denominator.

This invariant structure ensures several properties. First, concentrated liquidity: the AMM provides liquidity exclusively within [α, β], eliminating capital inefficiency from providing liquidity at prices that may never manifest.

Second, configurable liquidity distribution: the rotation angle determines where liquidity is deepest (at the peg price tan(φ)), while the stretch factor controls how sharply liquidity falls off away from the peg.

Third, swap invariance: valid trades preserve r (or increase it when fees are retained). Fourth, linearity in liquidity: proportional balance changes yield proportional invariant changes, enabling fungible LP shares where proportional deposits mint proportional BPT.

The mathematical foundation is based on the paper: E-CLP Mathematics by Klages-Mundt and Schuldenzucker.

Core Parts of GyroECLP Pool

Fixed-Point Arithmetic Foundation

To achieve sufficient numerical precision without excessive on-chain computation, the implementation uses dual-precision arithmetic : primary parameters (α, β, c, s, λ) are stored as 18-decimal fixed-point values and derived parameters including τ(α), τ(β) and auxiliary values $u,\ v,\ w,\ z,\ dSq$ are computed off-chain at 38-decimal precision , normalized on unit circle points for the price bounds and validated on-chain at deployment.

Components u, v, w, z are precomputed shortcuts for the transformed offset vector Aχ where (Aχ)_x = w/λ + z and (Aχ)_y = λu + v. The dSq parameter captures the actual value of c² + s² for precision correction.

Safety and Ratio Limits

Static limits (checked at pool creation):

`_ROTATION_VECTOR_NORM_ACCURACY= 1e3` enforces $|c^2 + s^2 - 1| \leq 1e^{-15}$.

`_MAX_STRETCH_FACTOR = 1e26` bounds $\lambda \leq 1e^{8}$.

`_DERIVED_TAU_NORM_ACCURACY_XP = 1e23` ensures $|\tau|^2 \approx 1$ within $1e^{-15}$

`_MAX_INV_INVARIANT_DENOMINATOR_XP = 1e43` prevents division instability when $(A\chi)^2 \approx 1$.

Dynamic limits (checked during operations): `_MAX_BALANCES = 1e34` ensures $x + y \leq 1e^{16}$ in normal precision, preventing overflow in squared terms.

`MAX_INVARIANT = 3e37` bounds $r \leq 3e19$ in normal precision.

Invariant ratio bounds: `MIN_INVARIANT_RATIO = 60e16` (60%) and `MAX_INVARIANT_RATIO = 500e16` (500%) constrain non-proportional liquidity operations to numerically stable regions.

Computing the invariant

The invariant r represents the radius of the underlying circle in the transformed coordinate system.

Given real reserves $t = (x,\ y)$, the invariant uniquely determines the pool's liquidity state: larger r corresponds to deeper liquidity, and proportional changes in reserves yield proportional changes in r.

The E-CLP uses a closed-form analytic solution derived from solving the ellipse equation for r. This eliminates iteration gas costs and convergence risks but requires explicit error bounding to handle the wide dynamic range of λ and the compounding effects of multiple fixed-point operations.

The implementation in `calculateInvariantWithError` (GyroECLPMath.sol) returns a tuple `(int256 invariant, int256 err)` rather than a single rounded value. This design allows callers to construct rigorous one-sided bounds: `invariant - err` provides an underestimate of the true mathematical value, while `invariant + err` provides an overestimate. The pool interface exposes them through `computeInvariant(ROUND_DOWN)` returning `invariant - err` and `computeInvariant(ROUND_UP)` returning `invariant + err`.

The invariant computation implements Proposition 13 (Section 2.2.1) from the E-CLP mathematics paper. Starting from the fundamental ellipse equation $\|A(t - \mu)\|^2 = r^2$ and substituting the offset relationship $\mu = r \cdot \chi$ yields a quadratic equation in r . The positive solution corresponding to the admissible arc of the ellipse is:

$$r = (At) \circ (A\chi) \pm \frac{\sqrt{[(At) \circ (A\chi)]^2 - [A\chi^2 - 1] \circ \|At\|^2}}{A\chi^2 - 1}$$

The transformed reserve vector At has components:

$$(At)_x = c \circ x - s \circ y \circ \lambda$$

$$(At)_y = s \circ x + c \circ y$$

where x,y are the pool balances, c,s are the rotation angle and λ = stretching factor.

The transformed offset vector Aχ has components computed from the derived parameters:

$$(A\chi)_x = \frac{w}{\lambda} + z$$

$$(A\chi)_y = \lambda \circ u + v$$

Where u, v, w, z are derived ECLP parameters stored in `DerivedEclpParams`.

The "+" solution is selected because it corresponds to reserves on the inner branch of the ellipse (the admissible trading arc). The "-" solution would place reserves on the outer branch, which lies outside the valid price range [α, β].

Implementation Steps

The calculation proceeds through four stages, each implemented as a separate function to manage complexity and enable precise error tracking.

Step 1 computes the dot product $At \circ A\chi$ via `calcAtAChi`. Expanding the component definitions:

$$At \circ A\chi = \frac{(cx - sy)(\frac{w}{\lambda} + z)}{\lambda \circ dSq^2} + \frac{(sx + cy)\,(\lambda u + v)}{dSq^2}$$

The division by dSq² accounts for four factors of the rotation components (s, c) in the product, correcting for any deviation of $s^2 + c^2$ from unity. The function rounds down in magnitude (toward zero) to produce an underestimate, the resulting error is compensated in later stages.

Step 2 computes the discriminant and its square root via `calcInvariantSqrt`. The discriminant: $D = [(At) \circ (A\chi)]^2 - [(A\chi)^2 - 1] \circ ||At||^2$ is decomposed into three terms computed by separate functions.

1. The function `calcMinAtxAChiySqPlusAtxSq` computes $-(At)\_x^2 \cdot (A\chi)\_y^2 + (At)\_x^2$.
2. The function `calc2AtxAtyAChixAChiy` computes $2 \cdot (At)\_x \cdot (At)\_y \cdot (A\chi)\_x \cdot (A\chi)\_y$.
3. The function `calcMinAtyAChixSqPlusAtySq` computes $-(At)\_y^2 \cdot (A\chi)\_x^2 + (At)\_y^2$.

Each term involves products of four rotation components, requiring division by dSq^4 for normalization. The terms are computed with sign-dependent rounding ($down$ for positive contributions, up for negative) to produce a conservative underestimate of the discriminant.

Step 3 computes the denominator term $(A\chi)^2 - 1$ via `calcAChiAChiInXp`. The squared magnitude expands to:

$$(A\chi)^2 = \frac{2\lambda uv}{dSq^3} + \frac{(u{+}1)^2 \circ \lambda^2}{dSq^3} + \frac{v^2}{dSq^3} + \frac{(w/\lambda + z)^2}{dSq^3}$$

This computation is performed entirely in extra precision (38 decimals). The $(u + 1)^2$ formulation rather than u^2 accounts for potential rounding errors.

The result is designed to exceed 1 for all valid pool configurations, ensuring the denominator remains positive. Rather than dividing by $(A\chi)^2 - 1$ directly, the implementation computes the reciprocal : $mulDenominator = (\frac{1e38}{(A\chi)^2 - 1}) - 1$ using `calcAChiAChiInXp` and multiplies it in subsequent steps, converting a division (which could overflow) into a multiplication with controlled precision.

Step 4 assembles the final invariant by combining the numerator terms and applying the denominator multiplier: `int256 invariant = (atAChi + sqrt - err).mulDownXpToNpU(mulDenominator);`

The subtraction of `err` from the numerator before multiplication ensures the final result is an underestimate. The `mulDownXpToNpU` function multiplies the extra-precision numerator by the extra-precision denominator reciprocal and converts the result to normal precision with downward rounding.

Error Tracking

The E-CLP tracks numerical error explicitly rather than relying on consistent directional rounding. Each stage of the calculation produces not just a result but also a bound on the maximum deviation from the true mathematical value.

The error inside the square root argument is dominated by extra-precision truncation effects. The initial bound is computed as: `err = (x.mulUpMagU(x) + y.mulUpMagU(y)) / 1e38;` which captures the $O((x^2 + y^2) / 10^{38}$ error from fixed-point arithmetic in the discriminant terms.

The square root error depends on the magnitude of the result. When $sqrt > 0$, error propagates according to the derivative of the square root function: `err = (err + 1).divUpMagU(2 * sqrt);` . When $sqrt \approx 0$ due to rounding artifacts driving the discriminant negative, the implementation sets `err = sqrt(err)` or a minimum of $1e^9$as a conservative fallback that avoids division by a near-zero value.

The numerator error combines the square root error with contributions from lambda-scaled balance terms: `err = ((lambda * (x + y) / 1e38) + err + 1) * 20;` The factor of 20 is a deliberate safety margin ensuring all possible error sources including ignored lower-order terms and normal-precision (10^{-18}) effects are captured within the bound.

The denominator error propagates through the multiplication with `mulDenominator`: `err = err.mulUpXpToNpU(mulDenominator);` . Since both the invariant and its error are scaled by the same denominator factor, small denominators amplify both proportionally. The error bound remains valid because the amplification affects the bound equally.

Precision and Overflow Protection

Multiple mechanisms prevent numerical failures. The balance constraint `x + y <= _MAX_BALANCES (1e34)` keeps squared terms within 256-bit bounds. The stretch factor bound `lambda <= _MAX_STRETCH_FACTOR (1e26)` limits λ^2 to 10^{52} in raw form, fitting safely in extra-precision representation. The invariant constraint `invariant + err <= _MAX_INVARIANT (3e37)` caps results to ensure dependent calculations remain safe.

Denominator stability is enforced by `_MAX_INV_INVARIANT_DENOMINATOR_XP = 1e43`, bounding the reciprocal $\frac{1}{[(A\chi)^2 - 1]}$ to at most 10^5 in extra precision. This prevents catastrophic amplification when $(A\chi)^2$ approaches 1. Pool creation validates this constraint via `validateDerivedParamsLimits`.

The discriminant is mathematically non-negative for valid states, but rounding errors can produce small negative values. The implementation treats these as zero: `val = val > 0 ? GyroPoolMath.sqrt(val.toUint256(), 5).toInt256() : int256(0);`

This is conservative: underestimating the square root term underestimates the invariant, maintaining the guarantee that `invariant <= r_true`. The square root uses Newton-Raphson iteration with seven steps and tolerance parameter 5, ensuring bounded approximation error.

Why This Is Safe

The explicit error tracking provides stronger guarantees than directional rounding alone. While directional rounding ensures each operation favors one side, accumulated errors across numerous operations can compound unpredictably.

The 20× and 40× safety multipliers are deliberately conservative, prioritizing safety margin over tight bounds. Parameter validation at deployment (`validateParams`, `validateDerivedParamsLimits`) eliminates pools with poorly-conditioned parameters—near-degenerate ellipses, extreme stretch factors, or malformed rotation vectors—before they can cause

Computing An Individual Balance

The `computeBalance` function in GyroECLPPool.sol determines the required balance of one token given the other token's balance and a target invariant ratio. This is used during non-proportional liquidity operations where the Vault needs to calculate how much of a token corresponds to a given change in pool liquidity.

Invariant Preparation

The function first computes the current invariant with error bounds as described in the previous section, then scales both bounds by the invariant ratio to obtain the target invariant:

```Rust
invariant.x = (currentInvariant + invErr).mulUp(invariantRatio) // overestimate
invariant.y = (currentInvariant - invErr).mulUp(invariantRatio) // underestimate
```

Both components use `mulUp` to be conservative. This dual-component vector allows downstream calculations to select whichever estimate produces a more conservative result based on the sign of intermediate terms. The function reverts if `invariant.x > _MAX_INVARIANT`.

Mathematical Formula

The balance computation implements Proposition 14 (Section 2.2.2) from the E-CLP mathematics paper. Given a known balance and target invariant r, the unknown balance is found by solving the ellipse equation. For computing y given x, the formula in shifted coordinates is:

$$y' = \frac{-s \cdot c \cdot \bar{\lambda} \cdot x' - \sqrt{D}}{1 - \bar{\lambda} \cdot s^2}$$

Where $\bar{\lambda} = 1 - 1/\lambda^2$, $x' = x - a$ and $y' = y - b$, the implementation evaluates the discriminant via the rearrangement (since $s^2 + c^2 = 1$) and makes it conservative by underestimating the first term and overestimating from :

$$D\_original = s^2 c^2 \bar{\lambda}^2 x'^2 - (1 - \bar{\lambda} s^2) \cdot [(1 - \bar{\lambda} c^2) x'^2 - r^2]$$

$$\text{To } D = (1 - \bar{\lambda} \cdot s^2) \cdot r^2 - \frac{x'^2}{y'^2}$$

The real reserve is recovered as $y = y' + b$. For computing x given y, the formula is symmetric with s ↔ c swapped and τ(β) replaced by -τ(α).

$$x' = \frac{-s \cdot c \cdot \bar{\lambda} \cdot y' - \sqrt{D}}{1 - \bar{\lambda} \cdot c^2}$$

The implementation computes an **overestimate** of the output balance through sign-aware rounding at each step.

The parameter $\bar{\lambda}$ is computed in both directions:

$$lamBar.x = 1 - 1/\lambda^2 \quad (rounded\ DOWN \rightarrow overestimate\ of\ \bar{\lambda})$$

$$lamBar.y = 1 - 1/\lambda^2 \quad (rounded\ UP \rightarrow underestimate\ of\ \bar{\lambda})$$

The numerator term $- sc\bar{\lambda}x'$ uses sign-dependent rounding when $x' > 0$ the negative product rounds toward zero (less negative), when $x' < 0$ the positive product rounds up. Both cases overestimate the numerator in signed direction.

The denominator $1 - \bar{\lambda} \cdot c^2$ is maintained as dual estimates: `sTerm.x` (overestimate) and `sTerm.y` (underestimate).

The discriminant D is deliberately underestimated by using $r.y$ (invariant underestimate) and the underestimate of $1 - \bar{\lambda} \cdot s^2$, while `calcXpXpDivLambdaLambda` computes x'^2/λ^2 as an overestimate using small padding constants (+3, +7) are added to extra precision terms to account for accumulated rounding errors.. A smaller discriminant produces a smaller square root, making the numerator larger (less negative) and thus overestimating y′ or x’.

The final division selects the appropriate denominator based on the sign of the numerator.

When positive, dividing by the underestimate `sTerm.y` yields a larger result. When negative, dividing by the overestimate `sTerm.x` produces a value closer to zero (larger in signed direction).

Virtual Offset Rounding

The offsets a and b are computed with sign-aware rounding to overestimate. For offset $a = \frac{r\cdot\lambda\cdot c\cdot\tau(\beta)_x}{dSq} + \frac{r\cdot s\cdot\tau(\beta)_y}{dSq}$, the implementation uses `r.x` (overestimate) with upward rounding when $\tau(\beta)_x > 0$, and `r.y` (underestimate) with downward rounding when $\tau(\beta)_x < 0$.

Why This Is Safe

The overestimation strategy ensures conservative outcomes for all liquidity operations. During single-sided additions, overestimating the required balance means users deposit more tokens than theoretically necessary. During single-sided removes, overestimating the remaining balance means users receive fewer tokens, preserving pool value.

The dual-component invariant vector propagates uncertainty through the quadratic formula by selecting the appropriate component at each step based on sign, achieving conservative rounding without explicit error tracking.

If rounding makes $D < 0$, the code clamps it to 0 before sqrt; since D is subtracted, this can only increase y′ which is conservative overestimate of the computed balance: a zero square root produces the smallest subtraction, yielding the largest output balance.

The denominator $1 - \bar{\lambda}\cdot s^2$ is guaranteed positive because $\bar{\lambda} < 1$ for all $\lambda \geq 1$ and $s^2 \leq 1$, so $\bar{\lambda}\cdot s^2 < 1$. Parameter bounds and dSq precision ensure rounding cannot push this negative.