Monolith Protocol: Technical Report

Date: December 2025

Solidity Version: 0.8.24

Framework: Foundry

License: UNLICENSED

MONOLITH

Audited by: rubencrxz

Table of Contents

1. Protocol Overview

Monolith is an **autonomous single-collateral CDP (Collateralized Debt Position) stablecoin factory** deployed on EVM-compatible networks. It allows any user or organization to deploy an independent, isolated stablecoin system by providing parameters to a shared `Factory` contract. Each deployment produces a fully self-contained triplet: a `Lender`, a `Coin` (the stablecoin), and a `Vault` (yield-bearing staking contract).

Protocol type: Collateralized Debt Position (CDP) / Algorithmic Stablecoin Factory **Peg mechanism:** Redemptions (hard peg), PSM buy/sell (soft peg), and interest rate adjustments **Interest bearing:** Yes — non-

redeemable borrowers pay a variable borrow rate **Governance:** Minimal — time-limited operator role, no on-chain voting

The protocol does not pool collateral across deployments. Each `Lender` instance is fully isolated, carrying its own collateral, debt, oracle, and parameters.

---

2. System Architecture

```
Factory (singleton)
│
├── InterestModel (singleton, shared by all Lenders)
│
└── Per-deployment triplet (deployed via CREATE3):
    ├── Lender      — Core CDP engine
    ├── Coin        — ERC20 stablecoin (18 decimals)
    └── Vault       — ERC4626 staking vault
```

Key relationships:

- `Factory` is the sole deployer and registry of `Lender` instances.
- `Lender` is the exclusive `minter` of its associated `Coin`.
- `Vault` holds `Coin` as its underlying asset and queries `Lender.getPendingInterest()` for real-time yield accounting.
- `InterestModel` is a stateless pure-math contract shared by all `Lender` instances; it is deployed once at `Factory` construction.
- `Lens` is a stateless read-only helper contract that simulates the `updateBorrower` logic off-chain without modifying state.

Data flow (borrow path):

```
User → Lender.adjust(+collateral, +debt)
         ├── collateral.safeTransferFrom(user → Lender)
         └── Coin.mint(user, debtAmount)
```

Data flow (repay path):

```
User → Lender.adjust(0, -debt)
         ├── Coin.transferFrom(user → Lender)
         └── Coin.burn(amount)
```

---

3. Contract Descriptions

3.1 `Factory.sol`

The `Factory` is a **singleton registry and deployer**. It:

- Deploys `Lender`, `Coin`, and `Vault` triplets via `CREATE3` using four deployer libraries (`InterestModelDeployer`, `LenderDeployer`, `VaultDeployer`, `CoinDeployer`).
- Maintains a registry (`deployments[]` array and `isDeployed` mapping).
- Controls global fee parameters (`feeBps`, `feeRecipient`) and per-lender custom fees (`customFeeBps`).
- Caps the global fee at `MAX_FEE_BPS = 1000` (10%).
- Owns `pullReserves()`, which allows `feeRecipient` to pull accrued global reserves from any registered `Lender`.
- Uses a two-step ownership transfer (`setPendingOperator` / `acceptOperator`).

Constructor parameters:

Parameter	Type	Description
`_operator`	`address`	Initial operator
`_minDebtFloor`	`uint256`	Minimum allowed `minDebt` for any deployment

Deployment parameters (`DeployParams`):

Parameter	Type	Description
`name`, `symbol`	`string`	Token name/symbol for Coin and Vault
`collateral`	`address`	ERC20 collateral token
`psmAsset`	`address`	PSM asset (optional, `address(0)` to disable)
`psmVault`	`address`	ERC4626 vault for PSM asset (optional)
`feed`	`address`	Chainlink-compatible price feed
`collateralFactor`	`uint256`	In bps, max 8500 (85%)
`minDebt`	`uint256`	Minimum position debt, ≥ `minDebtFloor`
`timeUntilImmutability`	`uint256`	Seconds until operator role expires, < 1460 days
`operator`	`address`	Per-lender operator
`manager`	`address`	Per-lender manager (subset of operator permissions)
`halfLife`	`uint64`	Interest rate half-life in seconds, [24h, 30d]
`targetFreeDebtRatioStartBps`	`uint16`	Start of target band, [500, endBps]
`targetFreeDebtRatioEndBps`	`uint16`	End of target band, [startBps, 9500]
`redeemFeeBps`	`uint16`	Fee charged on redemptions, ≤ 1000 (10%)
`stalenessThreshold`	`uint32`	Oracle staleness threshold in seconds
`maxBorrowDeltaBps`	`uint16`	Max rounding delta on borrow, [50, 200] bps
`minTotalSupply`	`uint128`	Min PSM vault total supply required to buy (PSM guard)

3.2 `Lender.sol`

The `Lender` is the **central protocol engine** per deployment. It manages:

- Position adjustments (collateral deposits/withdrawals, debt increases/decreases).
- Interest accrual on the paid debt pool.
- Liquidations and write-offs.
- Redemptions from the free debt pool.
- PSM buy/sell operations.
- Reserve accounting (local and global fees).
- Oracle price consumption with staleness handling.

State variable layout (two packed 256-bit slots):

Slot 1:

```
uint16  targetFreeDebtRatioStartBps
uint16  targetFreeDebtRatioEndBps
uint16  redeemFeeBps
uint64  expRate
uint40  lastAccrue
uint88  lastBorrowRateMantissa   (initial: 2e16 = 2% APR)
uint16  feeBps
```

Slot 2:

```
uint16  cachedGlobalFeeBps
uint120 accruedLocalReserves
uint120 accruedGlobalReserves
```

Key constants:

Constant	Value	Description
`STALENESS_UNWIND_DURATION`	24 hours	Duration over which stale price decays to zero
`MIN_LIQUIDATION_DEBT`	10,000e18	Minimum liquidation chunk in Coin units
`MAX_DECIMALS`	30	Maximum supported collateral token decimals
`INTEREST_CALCULATION_GAS_REQUIREMENT`	40,000	Minimum gas required for interest accrual
`WRITEOFF_GAS_REQUIREMENT`	120,000	Minimum gas required for write-off
`MIN_RATE` (InterestModel)	5e15	0.5% APR minimum borrow rate

3.3 `Coin.sol`

A minimal **ERC20 stablecoin** (18 decimals) with a single immutable `minter` address (the `Lender`).

- `mint(address to, uint amount)` — callable only by `minter`.
- `burn(uint amount)` — callable by any token holder (burns from `msg.sender`).

No permit, no admin, no upgradeability. The token's entire supply is controlled exclusively through the `Lender`.

3.4 `Vault.sol`

An **ERC4626-compliant yield-bearing vault** accepting `Coin` as its underlying asset. Stakers deposit `Coin` and receive share tokens (`s<SYMBOL>`).

Yield source: Interest accrued on `totalPaidDebt` (paid by non-redeemable borrowers), distributed net of fees.

`totalAssets()` implementation:

```
totalAssets = coin.balanceOf(address(vault)) + lender.getPendingInterest()
```

This forward-includes accrued-but-not-yet-minted interest, ensuring share price accuracy between accruals.

First deposit protection: On the first deposit, `MIN_SHARES = 1e16` (0.01 Coin) worth of shares are burned to `address(0)`, preventing share price manipulation via donation attacks (the dead shares pattern).

3.5 `InterestModel.sol`

A **stateless pure-math contract** shared by all `Lender` instances. Deployed once in `Factory`'s constructor. Its sole function is `calculateInterest()`.

The design rationale for external deployment (rather than internal library): allows `Lender` to use a `try/catch` around the call, so arithmetic failures in the interest model do not brick the `Lender`. In the event of a revert, `accrueInterest()` silently skips accrual while preserving the last known rate.

3.6 `Lens.sol`

A **stateless view-only helper** that simulates `Lender.updateBorrower()` off-chain without consuming gas from state mutation. It exposes:

- `getCollateralOf(lender, borrower)` — returns actual collateral balance accounting for pro-rata redemptions across epochs.
- `getDebtOf(lender, borrower)` — returns debt including accrued-but-not-committed interest, accounting for epoch share reductions.

---

4. Dual Debt System

The protocol distinguishes two mutually exclusive debt pools per `Lender`:

Attribute	Free Debt (Redeemable)	Paid Debt (Non-Redeemable)
Borrow rate	0%	Variable (≥ 0.5%)
Redemption exposure	Yes	No
Default status	No (opt-in)	Yes (default)
Debt tracking	`freeDebtShares`	`paidDebtShares`
Pool totals	`totalFreeDebt`, `totalFreeDebtShares`	`totalPaidDebt`, `totalPaidDebtShares`

Switching between pools is done via `setRedemptionStatus(account, chooseRedeemable)` or the overloaded `adjust(..., bool chooseRedeemable)`. The switch atomically moves debt between pools using a decrease-then-increase share operation, with a guard requiring `currDebt >= prevDebt` after the switch (rounding can only increase debt).

Debt accounting: Both pools use a shares/total model:

```
debtOf(account) = shares[account] * totalDebt / totalShares   (mulDivUp)
```

Interest accrual only updates `totalPaidDebt` (increasing shares dilute, i.e., all non-redeemable borrowers pay pro-rata interest). Free debt is never inflated by interest.

---

5. Interest Rate Model

Type: Exponential convergence (continuous-time, closed-form integral)

The rate target is defined by a band: `[targetFreeDebtRatioStartBps, targetFreeDebtRatioEndBps]`.

`getFreeDebtRatio()`:

```
freeDebtRatio = (totalFreeDebt + normalizePsmAssets(freePsmAssets))
                / (totalPaidDebt + totalFreeDebt +
normalizePsmAssets(freePsmAssets))
```

PSM assets are counted as free debt equivalent, since they represent Coin in circulation backed 1:1 and not subject to interest.

Rate update logic per accrual:

Condition	Rate behavior	Interest formula

Condition	Rate behavior	Interest formula
`freeDebtRatio < targetStart`	Rate grows: `rate * e^(expRate * Δt)`	`totalPaidDebt * (newRate - lastRate) / expRate / 365d`
`freeDebtRatio > targetEnd`	Rate decays: `rate * e^(-expRate * Δt)`	`totalPaidDebt * (lastRate - newRate) / expRate / 365d`
Within band	Rate constant	`totalPaidDebt * lastRate * Δt / 365d / 1e18`

The **closed-form integral** of the exponential curve is used for accurate interest computation (not a Riemann approximation), ensuring precision across large `Δt` gaps.

`expRate` **calculation:**

```
expRate = wadLn(2 * 1e18) / halfLife    (≈ 0.693147 / halfLife)
```

`halfLife` determines how quickly rates respond: smaller values mean faster convergence.

Floor: `MIN_RATE = 5e15` (0.5% APR). When decay would push the rate below this floor, the integral is split into a decaying portion and a flat portion at floor rate.

Overflow guards: If `interest > type(uint120).max` or `currBorrowRate > type(uint88).max`, the model returns `(lastRate, 0)` — skipping accrual rather than reverting.

---

6. Peg Stability Module (PSM)

The PSM is an **optional feature** enabled by setting `psmAsset` and/or `psmVault` at deployment time. It provides a direct 1:1 swap between the stablecoin (`Coin`) and a reference asset (e.g., USDC, USDT).

Two PSM configurations:

Config	`psmVault`	Behavior
Direct	`address(0)`	PSM holds raw `psmAsset` tokens
Yield-bearing	ERC4626 vault	PSM deposits `psmAsset` into `psmVault`, earns yield

`buy(assetIn, minCoinOut)` — User sends `psmAsset`, receives `Coin` 1:1 (decimal-adjusted).

- Only callable before `immutabilityDeadline`.
- A buy fee ramps from 0% to 1% (100 bps) linearly over the second half of the `timeUntilImmutability` period.
- `minTotalSupply` guard: prevents `buy` when PSM vault supply is dangerously low.

`sell(coinIn, minAssetOut)` — User burns `Coin`, receives `psmAsset` 1:1.

- No fee on sell.

- Uses `psmVault.convertToShares()` to calculate redemption shares, ensuring no vault rounding losses exceed `minAssetOut`.

PSM yield (`accruePsmProfit`): Yield earned by the PSM vault above `freePsmAssets` is captured as `accruedLocalReserves` on the next state-modifying call. Rebasing tokens held directly (no vault) are similarly captured.

`reapprovePsmVault()`: Restores `type(uint).max` approval to the PSM vault (callable before deadline, for non-standard approval resets).

---

7. Redemption System

Redemptions allow `Coin` holders to exchange `Coin` for collateral at oracle price minus a `redeemFeeBps` fee, repaying free-debt borrowers' obligations pro-rata.

Mechanics:

1. Caller provides `amountIn` Coin; protocol calculates:

   ```
   internalAmountOut = amountIn * 1e18 * (10000 - redeemFeeBps) / price / 10000
   ```

2. `totalFreeDebt -= amountIn` (direct reduction).
3. `epochRedeemedCollateral[epoch] += internalAmountOut * 1e36 / totalFreeDebtShares` — a per-share index accumulator.
4. Collateral is transferred to the redeemer.

Epoch system: The epoch mechanism handles share dilution when the ratio of `totalFreeDebtShares / totalFreeDebt` diverges. When this ratio exceeds `1e9`, a new epoch is triggered:

- `epoch` increments.
- `totalFreeDebtShares` is scaled down by `1e36`: `totalFreeDebtShares = totalFreeDebtShares * 1e18 / 1e36`.

This prevents the per-share index from overflowing. Free debt shares within an epoch are reduced by dividing by `1e36` at each epoch boundary (`divWadUp(1e36)`), with a special case: if the result is 1, it rounds to 0 (effectively eliminating dust shares).

Lazy borrower updates (`updateBorrower`): Borrower state is updated lazily (on the next interaction). Up to **5 epoch transitions** are processed per call. For each missed epoch:

- The borrower's collateral is reduced proportionally to their share of free debt redeemed.
- Their shares are reduced by `divWadUp(1e36)`.

`nonRedeemableCollateral` tracking: A global accumulator tracks total collateral belonging to non-redeemable borrowers. Redemptions verify:

```
totalInternalCollateral - internalAmountOut >= nonRedeemableCollateral
```

This ensures redemptions never seize non-redeemable collateral.

`getRedeemAmountOut()` constraints:

- Returns 0 if `amountIn > totalFreeDebt`.
- Returns 0 if `allowLiquidations == false` (oracle invalid or stale).

---

8. Liquidation and Bad Debt Management

8.1 Liquidations

Trigger condition: `debt > borrowingPower`, i.e.:

```
debt > price * collateralBalance * collateralFactor / 1e18 / 10000
```

Liquidatable amount: Fixed at 25% of total debt per call, with a floor of `MIN_LIQUIDATION_DEBT = 10,000 Coin` (or full debt if debt is below this floor).

Liquidation incentive (dynamic): The incentive scales linearly between 0% and 10% based on how far LTV exceeds `collateralFactor`:

```
ltvBps = debt * 10000 / collateralValue
incentiveRange = [collateralFactor, collateralFactor + 500]  (500 bps = 5%)
incentiveBps   = linear interpolation → [0, 1000]           (0% to 10%)
```

Process:

1. Repay `repayAmount` from the borrower's debt pool.
2. Transfer `collateralReward = repayAmount * (10000 + incentiveBps) / 10000 / price` to liquidator.
3. Attempt `writeOff()` via external `try/catch`.

8.2 Write-Offs

Trigger condition: `debt > collateralValue * 100` (i.e., debt exceeds 100× the value of remaining collateral).

Process:

1. Delete the borrower's entire debt position.
2. Redistribute the written-off debt proportionally between `totalFreeDebt` and `totalPaidDebt` pools based on their current relative sizes.
3. Zero the borrower's collateral balance.
4. Transfer remaining collateral to the caller (the liquidator).

Bad debt socialization: Write-offs do not burn the redistributed debt — they add it to the existing debt pools. This means all borrowers (both free and paid) absorb the bad debt pro-rata through share dilution.

Gas guards: `writeOff()` is called via `try this.writeOff(borrower, to) catch` from `liquidate()`. If the call fails and insufficient gas was provided, the outer function reverts with `"Not enough gas for writeOff"`. This pattern prevents the write-off path from silently suppressing liquidation failures due to out-of-gas conditions.

---

9. Vault and Yield Distribution

9.1 Yield Accounting

Interest accrued on `totalPaidDebt` is distributed as follows after deducting fees:

```
interestAfterFees = interest - localReserveFee - globalReserveFee

if (totalStaked < totalPaidDebt):
    stakedInterest = interestAfterFees * totalStaked / totalPaidDebt
    remainder → accruedLocalReserves
else:
    stakedInterest = interestAfterFees (all to vault)
```

The cap `totalStaked < totalPaidDebt` ensures the **supply rate never exceeds the borrow rate**: if only a fraction of paid debt is staked, stakers only receive the proportional interest.

9.2 Fee Layers

Fee type	Parameter	Cap	Destination
Global fee	`cachedGlobalFeeBps` (from Factory)	10%	`accruedGlobalReserves` → `feeRecipient`
Local fee	`feeBps`	10%	`accruedLocalReserves` → `operator`
Redeem fee	`redeemFeeBps`	10%	Stays with redeemable borrowers (reduces amount out)
Buy fee	Ramps 0→1%	100 bps	`accruedLocalReserves`

9.3 Reserve Extraction

- **`pullLocalReserves()`** — callable by `operator`. Mints `accruedLocalReserves` in Coin to operator, then calls `accruePsmProfit()` first.
- **`pullGlobalReserves(to)`** — callable only by `factory` (via `Factory.pullReserves()`). Mints `accruedGlobalReserves` to `feeRecipient`.

Both functions call `accrueInterest()` before extraction to ensure up-to-date accounting.

---

10. Oracle and Price Feed Integration

Interface: `IChainlinkFeed` (compatible with Chainlink `AggregatorV3Interface`).

Price normalization: Prices are normalized to `(36 - collateralDecimals)` decimals, ensuring that:

```
price * collateralBalance (18 decimals) / 1e18 = value (18 decimals)
```

Normalization formula:

```
if (feedDecimals + 18 <= 36):
    price = feedPrice * 10^(36 - 18 - feedDecimals)
else:
    price = feedPrice / 10^(feedDecimals + 18 - 36)
```

Negative price handling: A negative `feedPrice` (invalid Chainlink state) is converted to 0, triggering reduce-only mode.

`getCollateralPrice()` state machine:

Condition	`reduceOnly`	`allowLiquidations`	`price`
Feed reverts	`true`	`false`	`1` (division guard)
`feedPrice <= 0`	`true`	`false`	`1`
`timeElapsed <= stalenessThreshold`	`false`	`true`	Raw normalized
`timeElapsed ∈ (threshold, threshold + 24h)`	`true`	`true`	Linear decay to 0
`timeElapsed > threshold + 24h`	`true`	`false`	`1`

The **24-hour staleness unwind** provides a gradual price reduction during the grace period after an oracle goes stale, allowing partial unwinding of positions rather than an abrupt freeze. Liquidations remain enabled during this period.

External call isolation: `getCollateralPrice()` calls `this.getFeedPrice()` externally to catch all possible reverts (including non-existent feed addresses).

---

11. Access Control and Governance

11.1 Roles

Role	Scope	Capabilities
`Factory.operator`	Global	Set `feeRecipient`, `feeBps`, `customFeeBps`, transfer operator role

Role	Scope	Capabilities
`Factory.feeRecipient`	Global	Call `pullReserves()` on any registered lender
`Lender.operator`	Per-lender	All `onlyOperatorOrManager` functions + `setLocalReserveFeeBps`, `pullLocalReserves`, `enableImmutabilityNow`, operator transfer
`Lender.manager`	Per-lender	`setHalfLife`, `setTargetFreeDebtRatio`, `setRedeemFeeBps`, `setMaxBorrowDeltaBps`, `setManager`

11.2 Immutability Deadline

Each `Lender` has an `immutabilityDeadline` set at deployment:

```
immutabilityDeadline = block.timestamp + timeUntilImmutability
```

Maximum value: `block.timestamp + 1460 days` (4 years).

After the deadline, all `beforeDeadline` functions become permanently non-callable:

- `setHalfLife`, `setTargetFreeDebtRatio`, `setRedeemFeeBps`, `setMaxBorrowDeltaBps`
- `buy` (PSM)
- `reapprovePsmVault`

`enableImmutabilityNow()` allows the operator to accelerate the deadline to `block.timestamp`, immediately locking these parameters.

Fee switch exemption: `setLocalReserveFeeBps` is not subject to the immutability deadline and remains callable by the operator indefinitely.

11.3 Delegation

Any borrower can delegate position control (`adjust`, `setRedemptionStatus`) to another address via:

```
delegate(address delegatee, bool isDelegatee)
```

Delegation is stored in a `mapping(address => mapping(address => bool))`.

11.4 Two-Step Operator Transfer

Both `Factory` and `Lender` implement a two-step operator transfer pattern:

1. `setPendingOperator(newOperator)` — sets candidate.
2. `acceptOperator()` — callable only by `pendingOperator`, completes the transfer.

---

12. Decimal and Precision Handling

All collateral is stored in an **internal 18-decimal representation** (`_cachedCollateralBalances`), regardless of the token's native decimals.

Conversion functions:

```
collateralToInternal(amount):
    if decimals == 18: return amount
    if decimals >  18: return amount / 10^(decimals - 18)    // truncates
    if decimals <  18: return amount * 10^(18 - decimals)

internalToCollateral(amount):
    if decimals == 18: return amount
    if decimals >  18: return amount * 10^(decimals - 18)
    if decimals <  18: return amount / 10^(18 - decimals)    // truncates
```

All internal math operates at 18 decimals. Conversions to/from token decimals happen only at the transfer boundary (deposits and withdrawals). The maximum supported collateral decimals is `MAX_DECIMALS = 30`.

PSM asset normalization (`normalizePsmAssets`): PSM asset amounts are normalized to 18 decimals for accounting purposes (e.g., `getFreeDebtRatio()`).

Share arithmetic: Uses `solmate`'s `FixedPointMathLib`:

- `mulDivUp` for debt-increasing operations (conservative — borrowers cannot underpay).
- `mulDivDown` for debt-decreasing operations (conservative — system cannot over-credit).

---

13. Storage Layout and Gas Optimizations

13.1 Packed Storage Slots

The two tightly packed 256-bit slots in `Lender` reduce cold read costs for frequently co-accessed variables:

Slot 1 (256 bits): `targetFreeDebtRatioStartBps` (16) + `targetFreeDebtRatioEndBps` (16) + `redeemFeeBps` (16) + `expRate` (64) + `lastAccrue` (40) + `lastBorrowRateMantissa` (88) + `feeBps` (16) = 256 bits exactly.

Slot 2 (256 bits): `cachedGlobalFeeBps` (16) + `accruedLocalReserves` (120) + `accruedGlobalReserves` (120) = 256 bits exactly.

13.2 Cached Global Fee

`cachedGlobalFeeBps` is refreshed from `factory.getFeeOf(address(this))` on every `accrueInterest()` call. This avoids a cross-contract call on each interest computation path while staying up-to-date.

13.3 Immutables

Frequently-read values (`coin`, `collateral`, `psmAsset`, `psmVault`, `feed`, `vault`, `interestModel`, `factory`, `collateralFactor`, `minDebt`, `minDebtFloor`, `deployTimestamp`, `psmAssetDecimals`,

`collateralDecimals`, `stalenessThreshold`, `minTotalSupply`) are declared `immutable`, avoiding SLOAD costs.

13.4 InterestModel as Shared Contract

Deploying `InterestModel` once and sharing it across all `Lender` instances reduces per-deployment bytecode size and Factory deployment costs, while the external call enables `try/catch` isolation.

---

14. External Dependencies

Library	Source	Usage
`solmate/ERC20`	Rari-Capital/solmate	`Coin` base, `collateral`, `psmAsset` interface
`solmate/ERC4626`	Rari-Capital/solmate	`Vault` base, `psmVault` interface
`solmate/SafeTransferLib`	Rari-Capital/solmate	Safe ERC20 transfers (no return value assumption)
`solmate/FixedPointMathLib`	Rari-Capital/solmate	`mulDivUp`, `mulDivDown`, `divWadUp`
`solmate/SignedWadMath`	Rari-Capital/solmate	`wadExp`, `wadLn` for interest rate exponentials
`solmate/CREATE3`	Rari-Capital/solmate	Deterministic address deployment

Chainlink: The protocol consumes a Chainlink-compatible `latestRoundData()` interface. No specific registry or feed validation beyond the `decimals()` and `latestRoundData()` function signatures is enforced at the contract level.

---

15. Deployment Architecture (CREATE3)

All three contracts in a deployment triplet (`Lender`, `Coin`, `Vault`) are deployed via `CREATE3`, providing **deterministic addresses independent of contract bytecode**.

Address derivation:

```
hash = keccak256(abi.encode("<type>", block.chainid, address(libraryDeployer),
caller, nonce))
address = CREATE3.getDeployed(hash)
```

Where `nonce = deployments.length` at the time of deployment.

Key property: The `Lender`'s address is known before deployment, allowing `Coin` and `Vault` to be initialized with the `Lender` address as a constructor argument. All three contracts reference each other correctly at

construction time without any post-deployment initialization calls.

Cross-chain safety: The hash includes `block.chainid`, preventing address collisions across chains for the same caller+nonce combination.

---

16. Invariants and Protocol Constraints

The following invariants are enforced by the protocol's code:

Collateral Factor:

```
collateralFactor ∈ [0, 8500] bps
```

Debt bounds:

```
debtOf(account) == 0 OR debtOf(account) >= minDebt >= minDebtFloor
```

Solvency (post-borrow/collateral-withdrawal):

```
debtOf(account) <= price * collateralBalance[account] * collateralFactor / 1e18 /
10000
```

Redeemable collateral:

```
totalInternalCollateral - redeemAmount >= nonRedeemableCollateral
```

Borrow delta guard:

```
actualDebtIncrease <= amount * (10000 + maxBorrowDeltaBps) / 10000
```

Redemption status switch (debt monotonicity):

```
debtAfterSwitch >= debtBeforeSwitch
```

Fee caps:

- Global factory fee: ≤ 10% (1000 bps)
- Local reserve fee: ≤ 10% (1000 bps)
- Redeem fee: ≤ 10% (1000 bps)

- Buy fee: ≤ 1% (100 bps, ramp)

Interest rate bounds:

- `lastBorrowRateMantissa` stored in `uint88` (max ≈ 309,485,000% APR)
- Floor: 0.5% APR (`MIN_RATE = 5e15`)
- Initial rate: 2% APR (`lastBorrowRateMantissa = 2e16`)

Reduce-only mode: When `reduceOnly == true`, users can only decrease debt or increase collateral. New borrows are blocked. The `adjust()` function enforces this via `require(!reduceOnly, "Reduce only")` in the solvency check path.

Liquidation disable: When `allowLiquidations == false` (oracle reverts or price decays to zero), `liquidate()` and `writeOff()` both revert.

Vault minimum shares: `MIN_SHARES = 1e16` are permanently burned to `address(0)` on the first vault deposit, preventing inflation attacks on the ERC4626 share price.

---

17. My Findings

17.1 Interest Undercharged when Borrow Rate Decays past MIN_RATE Due to Unit Mismatch in Integral Calculation

Summary

`InterestModel.calculateInterest()` undercharges interest when the borrow rate decays below `MIN_RATE`. In the decay branch, the function attempts to compute the interest as the integral of a decaying rate until `MIN_RATE`, plus a flat-rate portion at `MIN_RATE`. However, the two components of the integral are expressed in different units (seconds vs wad·seconds) and are summed together before being scaled down. As a result, the decay portion of the integral is effectively divided by 1e18 twice.

This causes borrowers to be charged significantly less interest than intended once the rate crosses `MIN_RATE`, even though the rate evolution itself is otherwise correct.

Root Cause

In the decay branch where `_lastFreeDebtRatioBps > _targetFreeDebtRatioEndBps`, the contract handles the case where the decayed rate would fall below `MIN_RATE` as follows (see `Monolith/src/InterestModel.sol` L46–L51):

```
interest = _totalPaidDebt * (
    (_lastRate - MIN_RATE) / _expRate +
    MIN_RATE * (_timeElapsed - timeToMin)
) / 365 days / 1e18;
```

- `(_lastRate - MIN_RATE) / _expRate` evaluates to **seconds** (the integral of the exponential decay).
- `MIN_RATE * (_timeElapsed - timeToMin)` evaluates to **wad·seconds**.
- The final division by `1e18` assumes the entire expression is in wad·seconds, but only the second term is.

As a consequence, the decay portion of the integral is scaled down by an extra factor of 1e18, effectively discarding most of the interest accumulated before reaching `MIN_RATE`.

Internal Preconditions

- `_lastFreeDebtRatioBps > _targetFreeDebtRatioEndBps` (system is in the decay region).
- `_lastRate > MIN_RATE`.
- `_timeElapsed` is large enough such that the continuously decaying rate satisfies `_lastRate * exp(-_expRate * _timeElapsed) < MIN_RATE`.

These conditions are determined by protocol state and elapsed time.

External Preconditions

- The protocol accrues interest using `InterestModel.calculateInterest()` (via `Lender.accrueInterest()`).
- A period of time passes between accrual calls (a normal on-chain condition, especially in low-activity markets).

Exploit Path

This is an economic correctness issue:

1. The system enters the decay region.
2. The borrow rate starts above `MIN_RATE`.
3. Enough time passes for the exponential decay to reach `MIN_RATE`.
4. On the next accrual, the contract computes interest using the faulty integral.
5. The borrower is charged only a small fraction of the interest that should have accrued before reaching `MIN_RATE`.

Impact

Borrowers are charged significantly less interest than intended once the borrow rate decays past `MIN_RATE`. This breaks the economic assumptions of the interest model and reduces the protocol's expected interest accrual over time.

Proof of Concept

The following Foundry test demonstrates the issue by comparing the interest returned by the contract against an integral computed with consistent units. Add this test to `InterestModel.t.sol` and run it (using `--via-ir` if necessary):

```
function test_PoC() public {
    uint totalPaidDebt = 1e19;
    uint lastRate      = 2e17;        // 20% APR (wad)
    uint MIN_RATE      = 5e15;        // 0.5% APR (wad)
    uint halfLife      = 2 days;
    uint expRate       = uint(wadLn(2e18)) / halfLife; // wad/sec

    // compute exact crossing time for THESE parameters
    uint timeToMin = uint(-wadLn(int(MIN_RATE * 1e18 / lastRate))) / expRate;
```

```
    // choose a very small margin over the minimum time needed
    uint timeElapsed = timeToMin + 1 hours;

    uint lastFreeDebtRatioBps = 9000; // > 7500 => decay branch
    uint startBps = 2500;
    uint endBps   = 7500;

    (uint currBorrowRate, uint interest) =
        interestModel.calculateInterest(
            totalPaidDebt,
            lastRate,
            timeElapsed,
            expRate,
            lastFreeDebtRatioBps,
            startBps,
            endBps
        );

    uint integralDecay_wadSec = ((lastRate - MIN_RATE) * 1e18) / expRate;
    uint integralFlat_wadSec  = MIN_RATE * (timeElapsed - timeToMin);
    uint expectedInterest = totalPaidDebt * (integralDecay_wadSec +
integralFlat_wadSec) / 365 days / 1e18;

    // Verifications
    console.log("timeElapsed", timeElapsed);
    console.log("currBorrowRate", currBorrowRate);
    console.log("interest (contract)", interest);
    console.log("expectedInterest", expectedInterest);

    assertLt(interest, expectedInterest, "Interest is not undercharged");
}
```

Expected Logs:

```
Logs:
  timeElapsed: 923229
  currBorrowRate: 5000000000000000
  interest (contract): 5707762557077
  expectedInterest: 15420805459727599
```

The logged values show that the interest charged by the contract is significantly smaller than the mathematically correct value. In this scenario, the contract charges 5,707,762,557,077 units of interest, while the correct integral yields 15,420,805,459,727,599.

Mitigation

Ensure both parts of the integral are expressed in the same unit (wad·seconds) before scaling.

17.2 PSM Misaccounting with Fee-Charging ERC-4626 Vaults

Summary

Monolith's PSM feature supports ERC-4626 vault tokens, but `Lender.buy()` and `Lender.sell()` assume 1:1 accounting between the user's "PSM asset amount" and the vault's economically backing amount.

ERC-4626 allows vaults to charge fees on deposit/mint and/or withdraw/redeem, and requires preview functions to account for those fees. In particular, the EIP defines "fee" as an amount charged to the user and states that fees may exist for deposits/withdrawals/etc.

When such a fee-charging but standard ERC-4626 vault is used as `psmVault`, the current Monolith logic deterministically:

1. mints Coin that is not fully backed by PSM reserves (entry/deposit fee case), and
2. mis-accounts `freePsmAssets` in a way that can eventually revert sells even while the vault still holds redeemable value (exit/withdraw/redeem fee case).

This breaks a stated critical invariant: *"PSM reserves must always be sufficient to redeem all PSM-backed stablecoin supply."*

Standard compliance

ERC-4626 explicitly:

- Defines fees and states they may exist for deposits/withdrawals/etc.
- Requires `previewDeposit` / `previewMint` to be inclusive of deposit fees.
- Requires `previewWithdraw` / `previewRedeem` to be inclusive of withdrawal fees.
- States `convertToShares` must not be inclusive of any fees, so using it to estimate share requirements around fee-charging flows is explicitly unsafe.

OpenZeppelin's ERC-4626 documentation also explains how fee-charging vaults remain compliant, and that integrators must ensure previews and execution math account for fees (including withdrawal quotes needing to include fees).

So, an ERC-4626 that charges entry/exit fees is still standard, and Monolith's PSM integration must handle it correctly.

Root cause

1. `buy()` over-mints Coin when the ERC-4626 charges an entry/deposit fee

In `Lender.buy()` (when `psmVault != 0`) (see `Monolith/src/Lender.sol` L519–L541):

- The protocol mints `coinOut` based on `assetIn`.
- It then deposits `assetIn` into `psmVault`.
- If the vault takes an entry fee, the lender receives fewer shares / less redeemable value than implied by `assetIn`.

Consequence: Each `buy()` mints more Coin than the PSM position gains in redeemable value, creating an immediate and accumulating backing shortfall.

2. `sell()` uses fee-exclusive conversions and decrements `freePsmAssets` by a pre-fee target amount

In `Lender.sell()` (when `psmVault != 0`) (see `Monolith/src/Lender.sol` L497–L517), the protocol:

- Computes `assetOut` from `coinIn` via a decimals-only conversion.
- Computes `sharesOut = psmVault.convertToShares(assetOut)`.
- Calls `psmVault.redeem()`.
- If the vault charges a withdraw/redeem fee, redeeming `sharesOut` returns strictly less than the pre-fee `assetOut` target amount. Despite that, Monolith updates accounting as:

  `freePsmAssets -= assetOut`, instead of subtracting what was actually paid out.

Consequence: `freePsmAssets` is pushed downward by the fee delta on every sell. This drift is monotonic and cumulative; it is guaranteed to grow with usage, and it directly corrupts any logic that relies on `freePsmAssets`.

Impact

- **PSM-backed Coin becomes under-collateralized**
  With a fee-charging ERC-4626 vault, every `buy()` mints Coin based on gross `assetIn`, while the PSM backing increases only by net assets after the vault's entry fee. Therefore, the system necessarily ends up with PSM-minted Coin supply exceeding redeemable PSM reserves by the cumulative amount of entry fees paid into the vault. This directly violates the protocol's stated invariant that PSM reserves must be sufficient to redeem all PSM-backed supply.
- **`freePsmAssets` accounting becomes wrong and eventually blocks `sell()`**
  With a fee-charging ERC-4626 vault, every `sell()` decrements `freePsmAssets` by `assetOut` even though the vault only returns `assetOut - fee` to the user. Because the drift is strictly increasing with each sell, there exists a deterministic point (after enough volume) where `freePsmAssets < assetOut` for a valid sell request and the subtraction `freePsmAssets -= assetOut` reverts.

Exploit Path

1. A Monolith deployment is created with a standard ERC-4626 vault as `psmVault` that charges an entry fee (deposit/mint fee), which is explicitly allowed by ERC-4626.
2. Users call `buy()`: the vault takes fees, so the lender receives less redeemable value than `assetIn`, but Monolith mints Coin as if the full `assetIn` backs it.
3. The system's PSM reserve backing becomes strictly less than PSM-backed Coin minted.
4. Analogously, with exit fees, `sell()` calls mis-account `freePsmAssets`.

Recommended mitigation

Mitigate by making PSM accounting fee-aware and based on real changes, not on nominal-input-based:

- **On `buy()`:** Determine the actual increase in redeemable backing caused by the deposit (e.g., compare the lender's vault share balance before/after and translate that to redeemable assets using the fee-inclusive preview mechanics), and mint Coin based on that net backing increase, not on `assetIn`.
- **On `sell()`:** If the user is meant to receive an exact `assetOut`, use the fee-inclusive quoting path (ERC-4626 previews) so the shares burned reflect fees. Update `freePsmAssets` based on the actual

economic change (or recompute it from the lender’s remaining vault share balance using the appropriate preview function), so it can’t drift.
