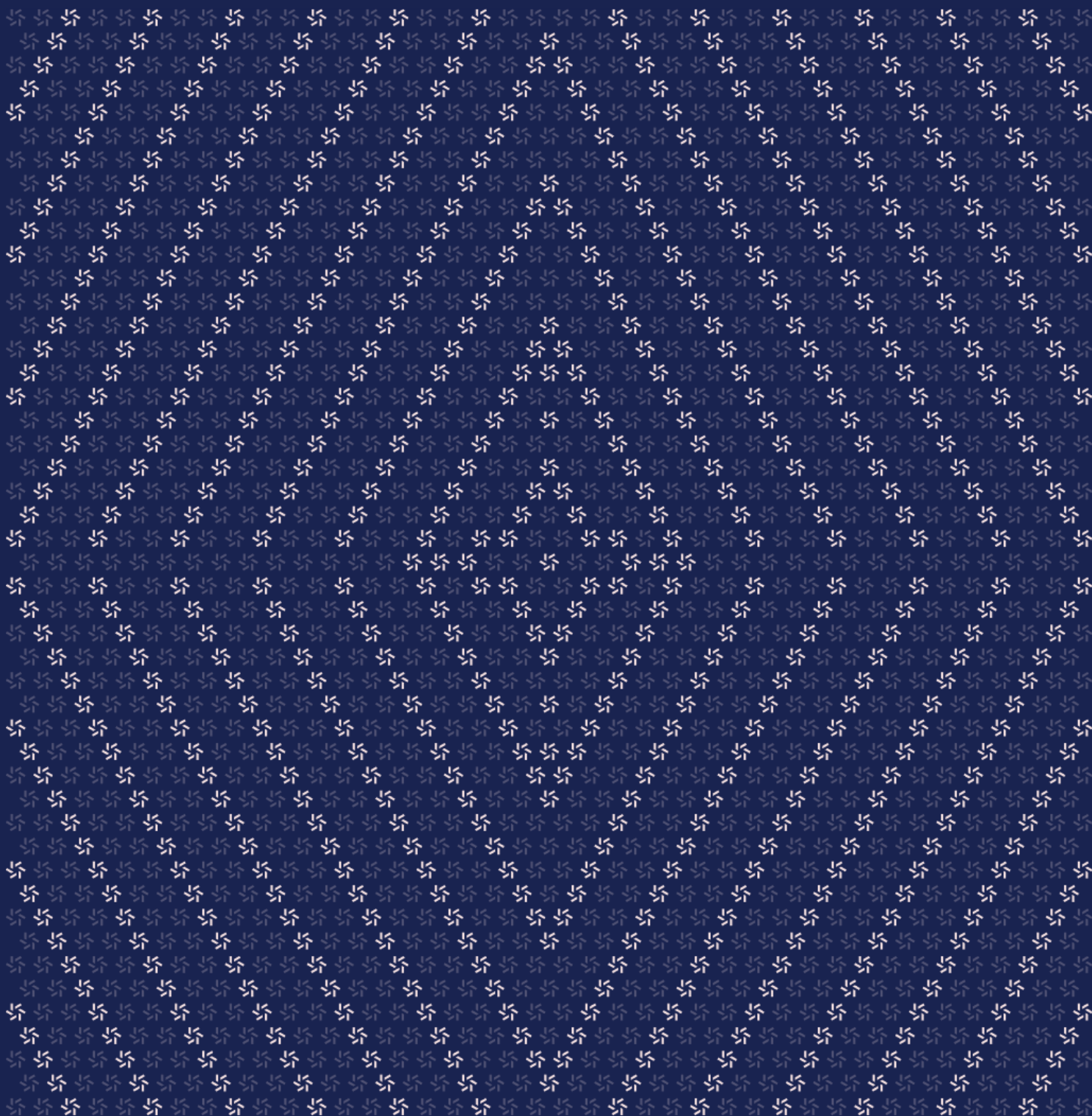


Jun 29, 2025

Metric OMM

Smart Contract Security Assessment



Contents

About Zellic	4
1. Overview	5
1.1. Executive Summary	5
1.2. Goals of the Assessment	5
1.3. Non-goals and Limitations	5
1.4. Results	6
2. Introduction	7
2.1. About Metric OMM	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	11
2.5. Project Timeline	12
3. Detailed Findings	13
3.1. PriceProviderL2 skips Chainlink validation during the sequencer grace period	13
3.2. Malicious pool may drain all tokens of user when called through router	15
3.3. Overly tight staleness window in PriceProviderL2 causes swap liveness risk	17
3.4. Zero-amount swap step can advance the active bin cursor without changing bin balances	19
3.5. Deployment process of a pool with a hook is not trivial	22

4.	Discussion	24
4.1.	Low-latency CEX-based oracle prices may expose LPs to temporary quote-manipulation risk	24
5.	Assessment Results	185
5.1.	Disclaimer	185

DRAFT

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io and follow [@zellic_io](https://twitter.com/zellic_io) on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io.



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Metric from May 20th to June 5th, 2026. During this engagement, Zellic reviewed Metric OMM's code for security vulnerabilities, design issues, and general weaknesses in security posture.

During the course of the audit, the codebase underwent several updates across multiple commits. The threat model described in this report is based on the following commit hashes:

- metric-core: 6aa6c3b489b84b8b1c50dc6a4967184df17aa395
 - metric-periphery: 0069084b927ebe891c6e7712c459ce4ac7e3118f
 - Oracle-Based-Pool: eda97bbcc1d33dd6c86f6993fa43d4e206755df0
-

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does Metric OMM correctly implement oracle-based pricing, bin-based liquidity accounting, drift limits, and swap math?
 - Could an on-chain attacker drain assets from Metric OMM pools or withdraw more liquidity than they are entitled to?
 - Could oracle, price-provider, or swap-reporting behavior be manipulated to create incorrect prices, unfair trades, or denial-of-service conditions?
 - Do the factory and periphery contracts correctly enforce access control, callback validation, slippage limits, deadlines, and native-token handling to prevent user fund loss or lockup?
-

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

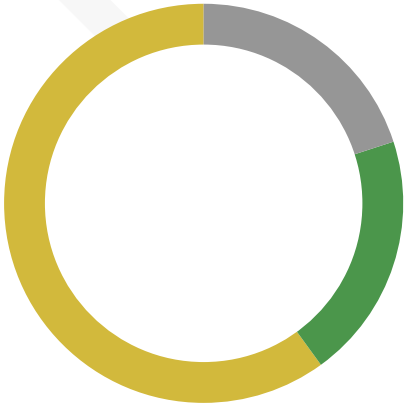
1.4. Results

During our assessment on the scoped Metric OMM contracts, we discovered five findings. No critical issues were found. Three findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Metric in the Discussion section (4. 7).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	1
Informational	1



2. Introduction

2.1. About Metric OMM

Metric contributed the following description of Metric OMM:

Metric is an oracle-guided liquidity venue for majors on EVM. Pools follow external reference pricing instead of discovering price inside reserves.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4.7) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

DRAFT

2.3. Scope

The engagement involved a review of the following targets:

Metric OMM Contracts

Type	Solidity
Platform	EVM-compatible
Target	metric-core
Repository	https://github.com/Metric-OMM/metric-core
Version	6aa6c3b489b84b8b1c50dc6a4967184df17aa395
Programs	Extload.sol MetricOmmPool.sol MetricOmmPoolDeployer.sol MetricOmmPoolFactory.sol libraries/BinDataLibrary.sol libraries/LiquidityLib.sol libraries/MetricHooks.sol libraries/PoolActions.sol libraries/PoolStateLibrary.sol libraries/SignedMath.sol libraries/Slot0Library.sol libraries/SwapMath.sol types/FactoryOperation.sol types/FactoryStorage.sol types/HookTypes.sol types/PoolOperation.sol types/PoolStorage.sol utils/MetricReentrancyGuardTransient.sol

Target	metric-periphery
Repository	https://github.com/Metric-OMM/metric-periphery
Version	0069084b927ebe891c6e7712c459ce4ac7e3118f
Programs	MetricOmmPoolLiquidityAdder.sol MetricOmmPoolSwapper.sol common/MetricOmmPoolQuoter.sol hooks/base/BaseMetricHook.sol hooks/base/SubhookUtils.sol hooks/examples/FullMetricHook.sol hooks/subhooks/DepositAllowlistSubhook.sol hooks/subhooks/OracleValueStopLossSubhook.sol hooks/subhooks/PriceVelocityGuardSubhook.sol hooks/subhooks/SwapAllowlistSubhook.sol hooks/subhooks/SwapReporterSubhook.sol lens/MetricOmmPoolDataProvider.sol lens/MetricOmmPoolStateView.sol

Target	smart-contracts-poc
Repository	https://github.com/Oracle-Based-Pool/smart-contracts-poc
Version	eda97bbcc1d33dd6c86f6993fa43d4e206755df0
Programs	PriceProvider.sol PriceProviderFactory.sol PriceProviderFactoryL2.sol PriceProviderL2.sol ProtectedPriceProvider.sol ProtectedPriceProviderL2.sol oracles/compressed/CompressedOracle.sol oracles/compressed/OracleBase.sol oracles/compressed/codebook/codebook.dull.sol oracles/providers/ChainlinkOracle.sol oracles/providers/OracleBase.sol oracles/providers/PythOracle.sol oracles/utils/ChainlinkVerifier.sol oracles/utils/ChainlinkVerifierL2.sol oracles/utils/Codebook256.sol oracles/utils/Extsload.sol oracles/utils/LazerConsumer.sol oracles/utils/OracleMath.sol oracles/utils/TimeMs.sol oracles/utils/U64x32.sol

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 5.2 person-weeks. The assessment was conducted by two consultants over the course of 2.4 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Hojung Han
Engineer
hojung@zellic.io

Jinseo Kim
Engineer
jinseo@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 20, 2026 Start of primary review period

May 22, 2026 Kick-off call

June 5, 2026 End of primary review period

3. Detailed Findings

3.1. PriceProviderL2 skips Chainlink validation during the sequencer grace period

Target	smart-contracts-poc/contracts/PriceProviderL2.sol		
Category	Coding Mistakes	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

PriceProviderL2 relies on an L2-specific Chainlink reference path as a sanity check for the primary oracle price. That Chainlink path is guarded by ChainlinkVerifierL2._checkSequencer(), which returns false when either:

- the L2 sequencer is down, or
- the L2 sequencer has recently recovered but is still inside the post-recovery grace period.

The issue is not that users can exploit the protocol while the sequencer is fully down. If the sequencer is down, users generally cannot submit L2 transactions. The relevant window is the post-recovery grace period, when transactions can resume but Chainlink's sequencer-uptime protection still indicates that Chainlink prices should not yet be trusted.

In the current flow, _cloPriceInQuote() handles a failed sequencer check by returning (false, 0, 0, 0). PriceProviderL2._getBidAndAskPrice() then treats this as the Chainlink reference price being unavailable and continues without performing the Chainlink deviation check.

As a result, during the sequencer grace period, PriceProviderL2 can still return usable bid/ask prices even though the sequencer protection mechanism has been triggered. This silently converts an explicit L2 safety signal into a best-effort Chainlink skip.

This finding is discovered in the commit b36d2275 of the smart-contracts-poc repository.

Impact

Price-dependent operations can continue during the post-recovery grace period without the intended Chainlink deviation validation. If the primary oracle is stale, delayed, or otherwise out of sync during that recovery window, the pool may price swaps using reduced safeguards despite the protocol having explicitly integrated L2 sequencer protection.

This weakens the intended fail-closed behavior of the sequencer-uptime check and can allow swaps to proceed during a period where Chainlink prices are intentionally considered unsafe to use.

Recommendations

Fail closed when the L2 sequencer check indicates that the sequencer is down or still inside the post-recovery grace period. In particular, `_closePriceInQuote()` should not return a generic “Chainlink unavailable” result for this condition if the caller will continue without the deviation check.

Consider separating Chainlink-reference states explicitly:

- Chainlink reference feed is not configured or intentionally optional;
- Chainlink reference data is unavailable for a non-safety-critical reason;
- L2 sequencer protection has triggered because the sequencer is down or within the grace period.

`PriceProviderL2._getBidAndAskPrice()` should revert, or otherwise return the same fail-closed sentinel used for unusable prices.

Remediation

This issue has been acknowledged by Metric, and a fix was implemented in commit [a65c82ae](#).

3.2. Malicious pool may drain all tokens of user when called through router

Target	MetricOmmPoolSwapper, MetricOmmPoolLiquidityAdder		
Category	Business Logic	Severity	High
Likelihood	Low	Impact	Medium

Description

MetricOmmPool invokes the caller's callback function as part of swapping tokens or adding liquidity to the pool. The callback function is responsible for paying the proper amount of tokens to the user.

MetricOmmPoolSwapper and MetricOmmPoolLiquidityAdder have the callback functions `metricOmmSwapCallback` and `metricOmmModifyLiquidityCallback`, respectively. These functions transfer the tokens by invoking `transferFrom` from the paying user to the pool.

These functions do not check whether the pool is "honest" (i.e., created by the factory), and they do not implement alternative checks on the input from the pool (e.g., that the callback is invoked with the correct amount, token, and swap direction, and only once). As a result, when a user swaps tokens or adds liquidity to the pool, the pool is effectively granted permission to indefinitely spend the router's allowances through the callback function.

*This finding is discovered in the commit **2b144c66** of the **metric-periphery** repository.*

Impact

If a user interacts (swapping tokens or adding liquidity) with a malicious pool through the router contracts (MetricOmmPoolSwapper or MetricOmmPoolLiquidityAdder), the malicious pool may drain all tokens the user approved to these router contracts.

Recommendations

Consider validating that the pool a user intends to transact with was deployed by the factory.

Remediation

This issue has been acknowledged by Metric.

Metric commented that this finding is only exploitable when a user explicitly swaps their tokens on a malicious pool or adds liquidity to a malicious pool, which would limit the impact of this issue.

We agree with this comment, while noting the caveat that the factory does not (and inherently cannot) provide a pool address for a given pair of tokens. On-chain contracts or dApp interfaces interacting with MetricOmmPoolSwapper or MetricOmmPoolLiquidityAdder should be aware of this issue and ensure that the pool contract provided by the untrusted party was deployed by the factory.

DRAFT

3.3. Overly tight staleness window in PriceProviderL2 causes swap liveness risk

Target	PriceProviderL2		
Category	Business Logic	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

PriceProviderL2._getBidAndAskPrice() validates price freshness using a fixed staleness threshold (MAX_TIME_DELTA) compared against the off-chain oracle's timestampMs field.

```
function _getBidAndAskPrice() internal view returns (uint128, uint128) {
    IOffchainOracle.OracleData memory data
    = offchainOracle.getOracleData(offchainFeedId);
    uint256 refTime = data.timestampMs.toSeconds();

    if (!_isStale(refTime, block.timestamp, MAX_TIME_DELTA)) {
        return (0, type(uint128).max);
    }
    ...
}
```

When the staleness check fails, the function returns (0, type(uint128).max), which causes MetricOmmPool._getBidAndAskPriceX64() to revert with BidIsZero, making all swaps unavailable until a fresh oracle update arrives.

On the currently deployed WETH/USDC pool on Ethereum Mainnet, MAX_TIME_DELTA is approximately 23 seconds — equivalent to roughly two 12-second blocks. This means if the oracle update transaction is not included in consecutive blocks, the swap functionality becomes unavailable even when the underlying market is operating normally.

This tightness creates a liveness dependency on consistent, low-latency oracle updates. The following real-world conditions can cause oracle updates to be delayed beyond 23 seconds:

- **Network congestion:** During periods of high gas demand (e.g., large NFT mints, airdrop claims, volatile market events), oracle keeper transactions may be outbid and delayed by several blocks.
- **Keeper infrastructure issues:** Any brief disruption to the oracle keeper's RPC node, transaction relay, or signing infrastructure can delay an update.
- **MEV / bundle competition:** Priority fee spikes can transiently exclude lower-priority transactions such as oracle updates.

Note that `simulateSwapAndRevert` does not enforce the `whenNotPaused` check and accepts caller-supplied bid/ask prices, so it remains available during staleness. However, all live swap calls are blocked for the duration of the stale window.

*This finding is discovered in the commit **b36d2275** of the **smart-contracts-poc** repository.*

Impact

Swap functionality is unavailable whenever the oracle has not been updated within `MAX_TIME_DELTA`. Under normal Ethereum Mainnet conditions, this can happen during any period of block space competition, causing a denial of service for traders with no mechanism for the pool itself to recover without an oracle update. LP positions remain accessible (add/remove liquidity is not gated by oracle freshness), but the pool's core trading utility is lost during these windows.

Recommendations

Consider one or more of the following mitigations:

1. **Increase `MAX_TIME_DELTA`** to a value that accommodates realistic oracle update latency under congested conditions (e.g., 60–120 seconds), accepting slightly wider price staleness tolerance in exchange for improved liveness.
2. **Use a graduated response:** Rather than immediately returning an invalid price pair (`0, type(uint128).max`) that causes a hard revert, consider reverting with a dedicated `StalePriceData` error that callers can distinguish from other oracle failures, or returning a configurable fallback spread that widens the bid/ask to discourage trading rather than halting it entirely.
3. **Monitor and alert:** If the tight threshold is intentional, ensure robust keeper monitoring with on-chain and off-chain alerting so that degraded liveness is detected and remediated quickly.
4. **Document the operational service-level agreement (SLA):** Clearly specify the expected oracle update frequency and the keeper infrastructure redundancy guarantees so that the liveness risk is understood by liquidity providers.

Remediation

This issue has been acknowledged by Metric. Metric notified us that the tight oracle staleness window is intentional because pool health depends primarily on update freshness, and they accept the resulting swap-liveness trade-off to protect liquidity from toxic flow.

3.4. Zero-amount swap step can advance the active bin cursor without changing bin balances

Target	SwapMath		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

`SwapMath.buyToken1InBinSpecifiedOut()` and `SwapMath.buyToken0InBinSpecifiedOut()` compute the next cursor position (`finalBinPos`) and the corresponding token amounts exchanged for a single bin during an exact-output swap. The `priceLimitX64` parameter restricts how far the price may move within the active bin.

When `priceLimitX64` clamps `finalBinPos` before any output can be produced — for example, when the price limit falls between `currBinPos` and the natural target — the functions may return a non-zero `finalBinPos` with `amountOutScaled == 0`, `amountInScaled == 0`, and zero fees. In this situation the bin state (`token0BalanceScaled`, `token1BalanceScaled`) is left unchanged, but `finalBinPos` is still returned as the price-limit-clamped value.

The pool's swap orchestration in `MetricOmmPool._finalizeSwap()` unconditionally commits this value:

```
function _finalizeSwap(int256 curBinIdxCache, uint256 curPosInBinCache,
    int256 curBinDistE6Cache) internal {
    curBinIdx = curBinIdxCache.toInt8();
    curPosInBin = curPosInBinCache.toUint104();
    curBinDistFromProvidedPriceE6 = curBinDistE6Cache.toInt24();
}
```

As a result, `curPosInBin` can change even though no tokens were exchanged and `binState.token0BalanceScaled / binState.token1BalanceScaled` remain identical. The active-bin cursor drifts relative to the actual distribution of liquidity within the bin.

Consider the following scenario in `buyToken0InBinSpecifiedOut`:

- `currBinPos` is at some mid-bin position; `binState.token0BalanceScaled` is very small (near zero but nonzero).
- `priceLimitX64` is set such that `finalBinPosAtPriceLimit < finalBinPos` (price limit clamps movement).
- `calculateOutputToken0FromBinPosition(binState.token0BalanceScaled,`

`currBinPos`, `finalBinPos`) rounds down to 0 because the balance is too small relative to the position movement.

4. `amountOutScaled` is set to 0, but `finalBinPos` is returned as the price-limit-clamped value rather than being reset to `currBinPos`.
5. The pool commits `finalBinPos` as the new `curPosInBin` without a corresponding change to bin token balances.

The analogous path exists in `buyToken1InBinSpecifiedOut` for the sell direction.

Impact

The cursor drift is bounded by the movement that `priceLimitX64` allows within a single bin step, and it requires the active bin to have an extremely small token balance on the output side to produce zero output at the clamped position. For the currently deployed pools with realistic liquidity levels, no profitable exploit scenario has been identified. However, the inconsistency between `curPosInBin` and `binState` balances introduces subtle accounting risk:

- Subsequent price calculations using `curPosInBin` (e.g., `calculatePriceAtBinPosition` in `_getInitialStateForSwap`) will reflect the drifted cursor, potentially producing a slightly different entry price for the next swap even though bin liquidity did not change.
- Over many repeated zero-output steps, the cursor could drift to `type(uint104).max` (triggering a bin traversal) or to 0 (triggering a downward bin traversal) without any real liquidity being consumed, though this requires an adversarially crafted sequence of price-limited calls in a very thin bin.

This finding is discovered in the commit [6aa6c3b4](#) of the `metric-core` repository.

Recommendations

Add an explicit guard in both `buyToken0InBinSpecifiedOut` and `buyToken1InBinSpecifiedOut` to reset `finalBinPos` to `currBinPos` when the resulting output amount is zero:

```
// After clamping finalBinPos by priceLimitX64:
if (amountOutScaled == 0) {
    finalBinPos = currBinPos;
    // delta0Scaled, delta1Scaled, binLpFeeAmount remain 0
    return (finalBinPos, delta0Scaled, delta1Scaled, binLpFeeAmount);
}
```

This ensures the invariant: *the cursor only moves if tokens are actually exchanged*.

The analogous fix should be applied consistently to both exact-output functions and verified for the exact-input counterparts (`buyToken0InBinSpecifiedIn`, `buyToken1InBinSpecifiedIn`) to

confirm the same guard is not needed there.

Remediation

This issue has been acknowledged by Metric, and a fix was implemented in commit [57674bdd](#).

DRAFT

3.5. Deployment process of a pool with a hook is not trivial

Target	BaseMetricHook, MetricOmmPool		
Category	Business Logic	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The constructor of BaseMetricHook requires the pool address, and the constructor of MetricOmmPool requires the hook address:

```
abstract contract BaseMetricHook is IMetricOmmHooks {
    // (...)
    constructor(address pool_) {
        pool = pool_;
    }
    // (...)
}
```

```
contract MetricOmmPool is IMetricOmmPool, Extstorage, ReentrancyGuardTransient {
    // (...)
    constructor(
        // (...)
    ) {
        // (...)
        HOOKS = hooks_;
        // (...)
    }
    // (...)
}
```

The test of FullMetricHook (which is based on BaseMetricHook) deploys itself and the pool connected to itself by (1) predicting the address of MetricOmmPool to be deployed through CREATE, (2) deploying the hook based on that pool address, and (3) deploying the pool with that hook. This method is not applicable in the production setting where the MetricOmmPool is deployed through the factory/deployer with CREATE2, because the hook address in the constructor parameter would affect the contract address.

*This finding is discovered in the commit **2b144c66** of the **metric-periphery** repository and the commit **ff639185** of the **metric-core** repository.*

Impact

This structure makes it not trivial to deploy the pool and hook.

Technically, it is not impossible to deploy the pool and hook in this structure. A deployer can (1) predict the address of the hook to be deployed through CREATE, (2) deploy the pool based on that hook address, and (3) deploy the hook through CREATE with that pool address. However, we believe this is not expected to be the canonical way to deploy a pool with a hook.

Recommendations

Consider refactoring the constructor functions of the pool and hook to simplify the deployment process.

Remediation

This issue has been acknowledged by Metric, and a fix was implemented in commit [7783672e](#).

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Low-latency CEX-based oracle prices may expose LPs to temporary quote-manipulation risk

If a Metric pool is configured to directly reflect, or reflect with very low latency, CEX-derived bid/ask quotes from an oracle such as Pyth Lazer, LPs may be exposed not only to impermanent loss from normal external market movements but also to temporary oracle-quote distortion.

In particular, for pairs with thin CEX order books or limited depth, an attacker may be able to briefly move one side of the order book and cause that distortion to be reflected by a data provider such as Pyth, Chainlink, or another oracle source. The attacker could then push the updated price on-chain, execute a swap against the Metric pool at the distorted price, and realize profit on the opposite leg once the CEX quote reverts to its normal level. In this scenario, the attacker's profit may effectively be extracted from the pool LP inventory. This is closer to a separate oracle-manipulation or market-source-manipulation risk than to ordinary impermanent loss caused by genuine market movement.

The Metric team acknowledged that this attack path is possible in principle. However, the attacker would need to move the external market or venue, ensure that the manipulated quote is incorporated by the data provider, and then submit the corresponding on-chain price update. This introduces both execution risk and an external cost of attack. The team also explained that Metric is intentionally price-source agnostic: the protocol does not mandate a specific oracle construction, and curators or integrators are expected to manage this risk through the design of their own price providers.

The Metric team considers this a significant risk if the price-provider logic is configured incorrectly, and plans to recommend conservative oracle configurations such as the following:

- Use native oracle confidence, volatility, or quality signals where available. For example, fast manipulation attempts should increase the effective confidence interval or spread applied by the price provider.
- Use an internal TWAP, a proprietary oracle, a secondary venue, or a Uniswap-style static AMM price/TWAP as a sanity-check source.
- Use a velocity-control hook to restrict trading when the referenced price moves too quickly.
- Use a stop-loss hook to limit the total amount of liquidity that can be drained during an attack, for example to a configured 1–2% loss threshold.
- Use an afterSwap hook to report historical trade information back to the price provider, allowing the price provider to incorporate pool trading history into its own logic.
- Treat Pyth and Chainlink as rigid sanity gatekeepers, while allowing curator-controlled price feeds to determine the final pool spread and quote more conservatively. For

example, a pool could require at least a 1–3 bps spread from the Chainlink/Pyth reference price and apply the most trader-unfavorable price when abnormal conditions are detected.

Accordingly, this risk may be disclosed as part of the broader LP risk model, but it should not be treated solely as ordinary impermanent loss. It should be explicitly documented as an oracle-manipulation or market-source-manipulation risk that must be bounded through conservative configuration. Curators should consider the listed asset's CEX depth, oracle update latency, confidence/spread handling, sanity-check venue, velocity limits, stop-loss limits, and pool liquidity caps so that the cost of manipulating the reference market remains higher than the value that could be extracted from LPs.

The Metric team also noted that a large and sophisticated market maker may be able to perform a broader one-sided inventory attack by accumulating an asset, moving the overall market price by some percentage, and then trading back against Metric liquidity. This structural risk cannot be fully eliminated. Metric's assumption is that, with appropriate liquidity caps, conservative spreads, sanity checks, and loss-limiting hooks, the cost of executing such an attack should exceed the potential gain from the pool's available liquidity.

Function: `function registerFeeds(bytes32[] calldata dsFeedIds, address[] calldata baseTokens, address[] calldata quoteTokens) external onlyRole(REGISTRAR_ROLE)`

`registerFeeds` is the function by which the registrar batch-registers multiple Data Streams feed IDs and their corresponding token pairs. It follows the same structure as the identically named function in `PythOracle`: all three arrays must have the same length, and each entry is passed to `_registerFeed` to add the feed ID to the registry and store base/quote token metadata. Unlike `PythOracle`, there is no `LazerIdOverflow` (uint32 range) check — Data Streams feed IDs use the full 32 bytes.

Inputs

- `dsFeedIds`
 - **Control:** `REGISTRAR_ROLE` Only
 - **Constraints:** All three arrays must have the same length (otherwise `LengthMismatch`). Each feed ID is added via `_addFeedId`; duplicates result in `FeedAlreadyExists`.
 - **Impact:** The Data Streams feed identifier. The upper nibble / schema version determines the decode branch in `updateReport`.
- `baseTokens`
 - **Control:** `REGISTRAR_ROLE` Only
 - **Constraints:** Must match the feed ID array in length. Each base token must

- be nonzero and differ from the quote token at the same index (`_setBaseAndQuoteTokens`).
 - **Impact:** The base asset for the provider.
- `quoteTokens`
 - **Control:** REGISTRAR_ROLE Only
 - **Constraints:** Must match the feed ID array in length. Each quote token must be nonzero and differ from the base token at the same index.
 - **Impact:** The quote asset for the provider.

Branches and code coverage

Intended branches

- If the caller has REGISTRAR_ROLE and all three arrays have the same length, the function registers each index via `_registerFeed`.
 - ☐ Test coverage
- When `_addFeedId` succeeds, `oracleData[...].timestampMs` for that feed is initialized to the sentinel `toTimeMs(1)`, so the first report must satisfy `isAfter(1ms)` (strict >) to be stored.
 - ☐ Test coverage
- If all three arrays have length 0, the loop does not execute and the function succeeds.
 - ☐ Test coverage

Negative branches

- If the caller does not have REGISTRAR_ROLE, `AccessControl` reverts.
 - ☐ Test coverage
- If the three array lengths do not match, the function reverts with `LengthMismatch()`.
 - ☐ Test coverage
- If `_addFeedId` returns false (duplicate) for any entry, the function reverts with `FeedAlreadyExists(feedId)`, rolling back the entire batch.
 - ☐ Test coverage
- If any entry has a zero or identical base/quote token, the require in `_setBaseAndQuoteTokens` fails, rolling back the entire batch.
 - ☐ Test coverage
- Base/quote token metadata is trusted registrar input only. The code verifies nonzero and distinct values but does not prove that the Data Streams feed semantically corresponds to the given pair or direction.
 - ☐ Test coverage

Function: `function registerFeed(bytes32 feedId, address baseToken, address quoteToken) public onlyRole(REGISTRAR_ROLE)`

`registerFeed` is the function that registers a single Data Streams feed. It calls `_registerFeed` to add the feed ID to the registry and set the base/quote tokens. This is the single-entry version of `registerFeeds` and applies the same validation.

Inputs

- `feedId`
 - **Control:** `REGISTRAR_ROLE` Only
 - **Constraints:** Added via `_addFeedId`; reverts with `FeedAlreadyExists(feedId)` if already present.
 - **Impact:** The Data Streams feed identifier to register.
- `baseToken`
 - **Control:** `REGISTRAR_ROLE` Only
 - **Constraints:** Must be nonzero and differ from `quoteToken`.
 - **Impact:** The base asset.
- `quoteToken`
 - **Control:** `REGISTRAR_ROLE` Only
 - **Constraints:** Must be nonzero and differ from `baseToken`.
 - **Impact:** The quote asset.

Branches and code coverage

Intended branches

- If the caller has `REGISTRAR_ROLE`, the feed ID is new, and the tokens are valid, the function registers the feed, sets `oracleData.timestampMs = toTimeMs(1)` as a sentinel, and emits `FeedRegistered`.

☒ Test coverage

Negative branches

- If the caller does not have `REGISTRAR_ROLE`, `AccessControl` reverts.

☒ Test coverage

- If `_addFeedId` returns false (duplicate), the function reverts with `FeedAlreadyExists(feedId)`.

☒ Test coverage

- If base or quote is zero or they are identical, the require in `_setBaseAndQuoteTokens` fails and the function reverts.

☐ Test coverage

- Because the function is public, it can also be called from within the contract, but in practice it is invoked by the registrar externally. At registration time, the guard/price-guard fields in the `FeedRegistered` event are emitted as placeholders (`0 / address(0)`).

☐ Test coverage

Function: `function updateReport(bytes calldata fullReport) external`

`updateReport` is a permissionless ingestion function that verifies a DON-signed Data Streams report via `VerifierProxy`, then decodes, normalizes, and stores it in the oracle storage. Rather than requiring permission, the authenticity of the data depends on DON signature verification by `verifierProxy.verify` (which reverts on failure). `_store` requires that the decoded feed ID is registered (`NotRegisteredFeedId`), that the timestamp is non-zero and within the future-drift limit, and that the timestamp is strictly newer (`isAfter`) than the existing value before storing.

Inputs

- `fullReport`
 - **Control:** Arbitrary
 - **Constraints:** Must pass DON signature and format verification by `verifierProxy.verify`. The decoded schema must be one of `v3`, `v4`, or `HFS` (otherwise `UnsupportedReportSchema`).
 - **Impact:** Verified and decoded, then stored as the price/spread/timestamp for the corresponding feed.

Branches and code coverage

Intended branches

- If the report is valid, the feed ID is registered, and the timestamp is the latest, the function updates `oracleData[feedId]` and emits `ReportStored`.

☒ Test coverage

- If the resolution nibble is `RESOLUTION_MILLISECONDS` (HFS), the function takes the `_normalizeHFS` path (timestamp is already in ms).

☒ Test coverage

- If the schema is `REPORT_V3`, the function uses `_normalizeV3` (spread is derived from bid/ask).
 - ☒ Test coverage
- If the schema is `REPORT_V4`, the function uses `_normalizeV4` (spread is 0 if marketStatus is `OPEN`, otherwise `BPS_BASE` as a stalled marker).
 - ☒ Test coverage
- If the decoded timestamp is the same as or earlier than the existing value (`!isAfter`), storage is skipped silently.
 - ☒ Test coverage

Negative branches

- If `verifierProxy.verify` fails (signature/format mismatch), the call reverts and nothing is stored.
 - ☐ Test coverage
- If the decoded schema is not `v3`, `v4`, or `HFS`, the function reverts with `UnsupportedReportSchema(schemaVersion)`.
 - ☒ Test coverage
- If the decoded feed ID is not registered, the function reverts with `NotRegisteredFeedId(feedId)`.
 - ☒ Test coverage
- If `d.timestampMs == 0`, the function reverts via `revertIfZero`.
 - ☒ Test coverage
- If `d.timestampMs` exceeds `block.time + MAX_TIME_DRIFT`, the function reverts via `revertIfAfterBlockTimeWithDrift`.
 - ☒ Test coverage
- If the report price is `<= 0`, the function reverts with `InvalidReportPrice` in `_toMid8/_spreadFromBidAsk`.
 - ☒ Test coverage
- If `ask < bid`, the function reverts with `InvalidReportPrice` in `_spreadFromBidAsk`.
 - ☐ Test coverage
- The function itself is permissionless, but `DON` signature verification acts as the gate. However, anyone can submit a verified report, so an attempt to update with an old — but within-future-drift — report is possible; the `isAfter` monotonicity check prevents stale overwrites.
 - ☐ Test coverage
- The 18-decimal Data Streams price is normalized to 8-decimal by dividing by `PRICE_SCALE(1e10)`, producing the same `OracleData` format as `PythOracle` so that the same `PriceProvider` can consume it directly.

- ☑ Test coverage
- VERIFICATION_FEE is currently the constant 0 wei, so no cost is deducted from the contract balance at present; however, since verify requires fee-token encoding, if the fee becomes non-zero in the future, the contract balance will need to be funded.
- ☑ Test coverage

Function call analysis

- `verifierProxy.verify{value: VERIFICATION_FEE}(fullReport, abi.encode(feeToken))`
 - **What is controllable?** `fullReport` is controlled by the caller. `feeToken` and `verifierProxy` are immutable.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned `reportData` is treated as a blob that has passed DON verification and is decoded. Arbitrary data that has not passed verification never reaches the return step because `verify` reverts.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If `verify` reverts, the entire ingestion reverts. `verifierProxy` is assumed to be the trusted Chainlink contract.

Function: `function updateReports(bytes[] calldata fullReports) external`

`updateReports` is the batch version of `updateReport` that iterates over multiple DON-signed reports and performs `_verifyReport` → `_store` for each. Each report is independently verified, normalized, and stored, and the same permissionless, DON-verification, registered-feed, and timestamp-monotonicity rules apply per entry.

Inputs

- `fullReports`
 - **Control:** Arbitrary
 - **Constraints:** Each element must pass `verifierProxy.verify` verification and the decoded schema must be v3, v4, or HFS.
 - **Impact:** Each entry is stored as the price/spread/timestamp for its corresponding feed.

Branches and code coverage

Intended branches

- If each report is valid, the feed is registered, and the timestamp is the latest, the function updates `oracleData` per entry and emits `ReportStored`.
 - ☒ Test coverage
- If the array length is 0, the loop does not execute and the function succeeds as a no-op.
 - ☐ Test coverage
- The function can handle a mix of v3, v4, and HFS schemas across entries.
 - ☐ Test coverage
- If a particular entry's timestamp is `!isAfter`, only that entry's storage is skipped and processing continues for the remaining entries.
 - ☐ Test coverage

Negative branches

- If `verifierProxy.verify` fails for any entry, the function reverts at that entry and the entire batch is rolled back.
 - ☐ Test coverage
- If the schema is unsupported for any entry, the function reverts with `UnsupportedReportSchema`, rolling back the entire batch.
 - ☐ Test coverage
- If the feed ID is not registered for any entry, the function reverts with `NotRegisteredFeedId(feedId)`, rolling back the entire batch.
 - ☐ Test coverage
- If any entry's timestamp is 0 or exceeds the future-drift limit, the function reverts via `revertIfZero/revertIfAfterBlockTimeWithDrift`.
 - ☐ Test coverage
- If any entry's price is `<= 0` or `ask < bid`, the function reverts with `InvalidReportPrice`.
 - ☐ Test coverage
- Because each entry's verification and storage occur atomically within a single transaction, if any entry fails, all previously stored entries in the batch are also rolled back. The batch is therefore all-or-nothing.
 - ☐ Test coverage

Function call analysis

- `verifierProxy.verify{value: VERIFICATION_FEE}(fullReports[i], abi.encode(feeToken))` (per entry, via `internal_verifyReport`)
 - **What is controllable?** Each `fullReports[i]` is controlled by the caller. `feeToken` and `verifierProxy` are immutable.

- **If the return value is controllable, how is it used and how can it go wrong?**
Each entry's returned `reportData` is treated as a blob that has passed DON verification and is decoded and stored. An entry that fails verification never reaches the return step and causes the batch to revert.
- **What happens if it reverts, reenters, or does other unusual control flow?** A `verifyRevert` for one entry rolls back the entire batch. If `VERIFICATION_FEE` becomes non-zero, fees are deducted from the balance once per entry.

Function: `function allowPushers(uint256 deadline, address[] calldata pushers, bytes[] memory signatures) external`

`allowPushers` is the function by which a creator registers a `pusher`→creator namespace remapping using ECDSA signatures from the pushers.

Inputs

- `deadline`
 - **Control:** Arbitrary
 - **Constraints:** `block.timestamp <= deadline` must hold; equality is permitted. No nonce is included.
 - **Impact:** Prevents the use of expired authorizations, but by design, replay of a valid pusher signature before expiry is permitted, allowing an already-approved pusher to be re-registered.
- `pushers`
 - **Control:** Arbitrary
 - **Constraints:** Must pass self-remapping, creator self-namespace remapping, and ECDSA signature recovery checks. The contract-pusher approval path is not available in this function.
 - **Impact:** Determines who can push slot updates to the creator namespace.
- `signatures`
 - **Control:** Arbitrary
 - **Constraints:** Must have the same length as `pushers`, and each signature must recover to the corresponding pusher address.
 - **Impact:** The authentication basis for EOA pusher delegation.

Branches and code coverage

Intended branches

- If `pushers.length == signatures.length`, the function matches each pusher with the signature at the same index and registers the creator namespace remapping.

☒ Test coverage

- If each signature recovers to the pusher address for the message (`chainid`, `contract`, `deadline`, `pusher`, `creator`), the function stores `namespaceRemapping[pusher] = msg.sender`.

☒ Test coverage

Negative branches

- If `block.timestamp > deadline`, the authorization has expired and the function reverts at the deadline check.

☐ Test coverage

- If `pushers.length != signatures.length`, the function cannot match pushers and signatures one-to-one and reverts.

☐ Test coverage

- If `namespaceRemapping[pusher] == pusher`, attempting to overwrite a namespace already set to the creator itself with a delegated pusher causes the function to revert with `CreatorCannotBeRemapped()`.

☐ Test coverage

- If `pusher == msg.sender`, attempting to register the creator itself as a delegated pusher causes the function to revert with `NoSelfRemapping()`.

☐ Test coverage

- If the signature recovery result does not match pusher, the pusher has not approved the delegation and the function reverts.

☐ Test coverage

- Because the authorization signature contains no nonce, an unexpired signature can be replayed to re-register a pusher even after a self-revoke, as long as the deadline has not passed.

☐ Test coverage

Function call analysis

- `ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(abi.encode(block.timestamp, address(this), deadline, pusher, msg.sender))), signatures[i])`
 - **What is controllable?** The signature and pusher/creator parameters are provided by the caller.
 - **If the return value is controllable, how is it used and how can it go wrong?** The recovered signer serves as the basis for pusher authorization.

- **What happens if it reverts, reenters, or does other unusual control flow?**
An invalid signature reverts, and no remapping occurs.

Function: `function allowContractPushers(uint256 deadline, address[] calldata pushers) external`

`allowContractPushers` is the function by which a creator registers a remapping based on the result of an `isPusher(creator)` staticcall to the pusher contract.

Inputs

- `deadline`
 - **Control:** Arbitrary
 - **Constraints:** `block.timestamp <= deadline` must hold; equality is permitted.
 - **Impact:** Prevents replay of expired authorizations or updates.
- `pushers`
 - **Control:** Arbitrary
 - **Constraints:** Must pass self-remapping, creator self-namespace remapping, and `isPusher(msg.sender)` staticcall approval checks. No signature verification is performed in this function.
 - **Impact:** Determines who can push slot updates to the creator namespace.

Branches and code coverage

Intended branches

- If each pusher contract's `isPusher(msg.sender)` staticcall succeeds and returns `true`, the function stores `namespaceRemapping[pusher] = msg.sender`.
☒ Test coverage
- If all pushers in the batch pass verification, the entire batch is registered in a single transaction.
☒ Test coverage

Negative branches

- If `block.timestamp > deadline`, the authorization has expired and the function reverts at the deadline check.
☒ Test coverage

- If `namespaceRemapping[pusher] == pusher`, attempting to overwrite a namespace already set to the creator itself with a delegated pusher causes the function to revert with `CreatorCannotBeRemapped()`.

☑ Test coverage

- If `pusher == msg.sender`, attempting to register the creator itself as a delegated pusher causes the function to revert with `NoSelfRemapping()`.

☑ Test coverage

- If `isPusher(msg.sender)` returns `false`, the caller is not the manager of that contract pusher and the function reverts.

☑ Test coverage

- `allowContractPushers` delegates the authorization decision to each caller-supplied pusher contract's `isPusher(address)` logic. No separate manager registry, factory check, or codehash allowlist is enforced here.

☑ Test coverage

Function call analysis

- `pusher.staticcall(abi.encodeWithSignature("isPusher(address)", msg.sender))`
 - **What is controllable?** The pusher contract address and the verification calldata input are provided by the caller.
 - **If the return value is controllable, how is it used and how can it go wrong?** The boolean response from `isPusher(msg.sender)` serves as the basis for permitting the pusher.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A failed `staticcall` or a false response must result in authorization failure.

Function: `function revokePusher()` external

`revokePusher` is the function by which a pusher removes its own creator remapping.

Branches and code coverage

Intended branches

- If the caller is a delegated pusher, the function clears `namespaceRemapping[msg.sender]` to zero.

☑ Test coverage

Negative branches

- If `namespaceRemapping[msg.sender] == address(0)`, the caller is not a registered pusher and the function reverts with `NoSelfRemapping()`.
☒ Test coverage
- If `namespaceRemapping[msg.sender] == msg.sender`, the caller is attempting to self-revoke the creator self-namespace and the function reverts with `NoSelfRemapping()`.
☒ Test coverage
- A self-revoke only clears the current mapping state. It does not prevent replay of existing unexpired `allowPushers` signatures issued by the creator.
☒ Test coverage

Function: `function removePushers(address[] calldata pushers) external`

`removePushers` is the function by which a creator removes a list of pushers that are mapped to it.

Inputs

- `pushers`
 - **Control:** Arbitrary
 - **Constraints:** Each pusher must satisfy `pusher != msg.sender` and `namespaceRemapping[pusher] == msg.sender`, meaning each listed pusher must currently point to `msg.sender` as its creator/manager. No signature or contract approval check is performed.
 - **Impact:** Removes the namespace update permission of delegated pushers mapped to the caller.

Branches and code coverage

Intended branches

- If each pusher is mapped to the caller creator, the function clears `namespaceRemapping[pusher]` to zero.
☒ Test coverage
- If all pushers in the array belong to the caller, the entire batch is removed in a single transaction.
☒ Test coverage

Negative branches

- If `pusher == msg.sender`, the creator is attempting to remove itself from remapping and the function reverts with `CreatorCannotBeRemapped()`.
☒ Test coverage
- If `namespaceRemapping[pusher] != msg.sender`, the caller is not the manager of that pusher and the function reverts with `InvalidManager(msg.sender)`, rolling back the entire batch.
☒ Test coverage

Function: `function createOracles(CreationRequest[] calldata requests) external override returns (CreationResponse[] memory res)`

`createOracles` is the function that creates compressed oracle feed registry entries using the caller's creator namespace. Rather than deploying a new contract, it initializes packed oracle data at creator-specific slots/positions and registers feed IDs, token metadata, guards, and occupancy.

Inputs

- `requests`
 - **Control:** Arbitrary
 - **Constraints:** An empty array returns an empty response. Each request must pass nonzero-token, base-not-equal-to-quote, `minPrice < maxPrice`, slot limit, and position limit checks.
 - **Impact:** Feed IDs, `oracleInfo`, `stateGuard`, `priceGuard`, slot occupancy, initial packed data, and creator registry pointers are set.

Branches and code coverage

Intended branches

- If `length == 0`, the function returns an empty `CreationResponse[]` without any state changes.
☐ Test coverage
- On first creation or creator namespace setup, `namespaceRemapping[msg.sender] = msg.sender` is set and the creator slot registry is used.
☒ Test coverage
- If a new slot starts (`r.nextPositionIndex == 0`), the new slot key is registered in `slotKeys`.

☒ Test coverage

- If a custom state guard is provided (`requests[i].stateGuard != address(0)`), that guard is used; otherwise the creator becomes the state guard.

☐ Test coverage

- If slot rollover occurs — that is, `r.nextPositionIndex` reaches the slot capacity — `currentSlotIndex` is incremented and the function moves to the next slot.

☒ Test coverage**Negative branches**

- `SlotLimitExceeded`: If the creator's `currentSlotIndex` is greater than or equal to `MAX_SLOTS_PER_CREATOR`, the function reverts.

☒ Test coverage

- `BadCreationRequest`: If the base or quote token is zero or the two tokens are identical, the function reverts.

☐ Test coverage

- `require(requests[i].minPrice < requests[i].maxPrice)`: If the price guard range is invalid, the function reverts.

☐ Test coverage

- If any request in the batch loop fails, all preceding feed creations are also rolled back as a single transaction unit.

☐ Test coverage

Function: `function updateBySignature(address feedCreator, uint256 deadline, uint256 newSlotValue, bytes calldata signature) external override returns (bool)`

`updateBySignature` is the function that updates a compressed oracle slot using a slot word signed by the feed creator.

Inputs

- `feedCreator`
 - **Control:** Arbitrary
 - **Constraints:** Must not be the zero address. There is no separate registry-exists check; the subsequent `slotId <= r.currentSlotIndex` and `_isDataValid` validations cover default registry values as well.
 - **Impact:** Determines which creator slot is updated.

- `deadline`
 - **Control:** Arbitrary
 - **Constraints:** `block.timestamp <= deadline` must hold; equality is permitted.
 - **Impact:** Prevents replay of expired authorizations or updates.
- `newSlotValue`
 - **Control:** Arbitrary
 - **Constraints:** Must pass future-drift timestamp check, newer-than-current-slot check, slot ID bound check, and initialized-position data validity check.
 - **Impact:** The actual stored value containing price data for multiple feeds.
- `signature`
 - **Control:** Arbitrary
 - **Constraints:** ECDSA recovery is used to verify the `feedCreator` signature only when the timestamp is newer than the current slot timestamp.
 - **Impact:** Affects the target of the function's state changes or external calls.

Branches and code coverage

Intended branches

- If the timestamp in `newSlotValue` is newer than the existing slot timestamp, the function verifies the signature and slot validity, updates the packed slot storage, and returns `true`.
 - ☒ Test coverage
- If the timestamp in `newSlotValue` is not newer than the existing value, the function returns `false` without modifying storage.
 - ☐ Test coverage

Negative branches

- If `block.timestamp > deadline`, the signed update has expired and the function reverts at the deadline check.
 - ☐ Test coverage
- If `feedCreator == address(0)`, a namespace cannot be created and the function reverts with `InvalidNamespace()`.
 - ☐ Test coverage
- When the timestamp is newer than the current slot timestamp, if the signature recovery result does not match `feedCreator`, the slot update is not approved by the creator and the function reverts. If the timestamp is not newer, the function returns `false` before signature verification.

☐ Test coverage

- If the timestamp in `newSlotValue` exceeds the permitted future drift (`MAX_TIME_DRIFT`), the function reverts in `revertIfAfterBlockTimeWithDrift`. This future-drift check is executed **before** the newer/stale comparison (`isAfter`) against the existing slot timestamp, and is independent of staleness, so an update exceeding the drift always reverts rather than returning `false`.

☐ Test coverage

- If `slotId > r.currentSlotIndex`, the function is attempting to update a slot not yet allocated to the creator and reverts with `InvalidSlotIndex()`.

☐ Test coverage

- If `_isDataValid(r, slotId, newSlotValue)` returns `false`, the data push is outside the range of initialized feed positions and the function reverts with `UnializedPush()`.

☒ Test coverage

- The signature is bound only to chain ID, contract, `feedCreator`, `deadline`, and `newSlotValue`, and is not bound to the feed ID, current slot layout, or layout nonce. Therefore, outstanding signatures may reference a stale layout after `moveFeeds/swapFeeds`.

☐ Test coverage

- The default registry value permits a timestamp-only write to slot 0 before initialization, allowing a valid signer to pre-seed the namespace timestamp before any feed is created. Subsequent updates must be strictly greater than that timestamp.

☐ Test coverage

Function call analysis

- `ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(abi.encode(block.timestamp, address(this), feedCreator, deadline, newSlotValue))), signature)`
 - **What is controllable?** `feedCreator`, `deadline`, `newSlotValue`, and `signature` are caller inputs.
 - **If the return value is controllable, how is it used and how can it go wrong?** The recovered signer must match `feedCreator` for the slot update to be permitted.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A malformed or invalid signature reverts, and no slot update occurs.

Function: `function deactivateFeeds(bytes32[] calldata feeds) external`

`deactivateFeeds` is a batch function by which a creator or admin resets the packed oracle data of existing feeds to default values and invalidates the `misc` pointer of each feed to `(address(0), 0, 0)`. Pointer invalidation is performed to prevent deactivated feeds from pointing to another feed's data after a subsequent `moveFeeds` reorganization.

Inputs

- `feeds`
 - **Control:** Creator/ADMIN_ROLE Only
 - **Constraints:** Each feed must exist in `feedIds`; the caller must be the feed creator or hold ADMIN_ROLE; and the current slot/position must be in the occupied state.
 - **Impact:** Each feed's packed data is reset to `DEFAULT_ORACLE_DATA_VALUE`, its occupancy bit is cleared, and `oracleInfo[feedId].misc` is invalidated.

Branches and code coverage

Intended branches

- If the array is empty, the loop does not execute and the function succeeds without any state changes.
 - ☒ Test coverage
- If the caller is the creator (`msg.sender == creator`), the feed owner can deactivate directly.
 - ☒ Test coverage
- If the caller holds ADMIN_ROLE (`hasRole(ADMIN_ROLE, msg.sender)`), the admin can deactivate on behalf of the creator.
 - ☒ Test coverage
- Due to batch atomicity, if any feed in the batch fails, the entire transaction reverts and all preceding deactivations are also rolled back.
 - ☒ Test coverage

Negative branches

- `require(feedIds.contains(feedId), FeedNotFound(feedId))`: An unregistered feed ID causes the function to revert.
 - ☒ Test coverage
- `require(msg.sender == creator || hasRole(ADMIN_ROLE, msg.sender), NotCreatorOrOwner())`: If the caller is neither a creator nor an admin, the function reverts.

☑ Test coverage

- `require(occ & (1 << pos) != 0, PositionEmpty(slotIndex, pos))`: If the position is already empty or invalidated, the function reverts.

☑ Test coverage

Function: `function moveFeeds(address creator, SlotMoveRequest[] call-data moves) external`

`moveFeeds` is a reorganization function by which a creator or admin moves a feed's slot/position to an empty target.

Inputs

- `creator`
 - **Control:** Creator/ADMIN_ROLE Only
 - **Constraints:** The caller must be the creator or an admin, and the namespace must be initialized.
 - **Impact:** Restricts the ownership scope of move/swap operations.
- `moves`
 - **Control:** Creator/ADMIN_ROLE Only
 - **Constraints:** Must pass source-occupied, target-free, and frontier-overwrite-prevention conditions.
 - **Impact:** Updates feed data positions and `misc` pointers.

Branches and code coverage

Intended branches

- If each move request is valid, the function copies the 48-bit oracle data from the source position to the target position and resets the source position to its default value.

☑ Test coverage
- If `oldSlot == newSlot`, the function clears the source bit and sets the target bit in the occupancy bitmap within the same slot.

☑ Test coverage
- If `oldSlot != newSlot`, the function updates the occupancy bitmaps of the source slot and target slot respectively.

☑ Test coverage

- If `!slotKeys.contains(newSlot)`, the target slot is added to the enumerable slot set so it is included in subsequent slot queries and management.

☑ Test coverage

Negative branches

- If the caller is neither creator nor a holder of `ADMIN_ROLE`, the function reverts with `NotCreatorOrOwner()`.

☑ Test coverage

- If `namespaceRemapping[creator] != creator`, the creator namespace is not active and the function reverts with `InvalidNamespace()`.

☑ Test coverage

- If the caller attempts to move a feed not in `feedIds`, the function reverts with `FeedNotFound(feedId)`.

☑ Test coverage

- If the packed misc creator of the feed differs from the requested creator, the function reverts with `FeedNotOwnedByCreator(feedId)`.

☑ Test coverage

- If `newPos >= MAX_PER_SLOT`, the target is outside the slot position range and the function reverts with `InvalidTargetPosition(newPos)`.

☑ Test coverage

- If `newSlotIdx > r.currentSlotIndex`, the function is attempting to move to an unallocated slot and reverts with `InvalidSlotIndex()`.

☑ Test coverage

- If `newSlotIdx == r.currentSlotIndex && newPos >= r.nextPositionIndex`, the move would overwrite a frontier position not yet allocated by `createOracles` and the function reverts with `InvalidTargetPosition(newPos)`.

☑ Test coverage

- If `oldSlotIdx == newSlotIdx && oldPos == newPos`, the move is a no-op to the same position and the function reverts with `MoveToSamePosition(feedId)`.

☑ Test coverage

- If the source occupancy has no `oldPos` bit set, there is no data to move and the function reverts with `PositionEmpty(oldSlotIdx, oldPos)`.

☑ Test coverage

- If the target occupancy already has the `newPos` bit set, overwriting it would corrupt another feed's data and the function reverts with `PositionOccupied(newSlotIdx, newPos)`.

☑ Test coverage

- Feed moves/swaps remap feed pointers but do not invalidate outstanding signed slot

words. Unless existing signatures are drained/expired or signed against a layout nonce, old-position data may be applied to the new feed mapping.

☒ Test coverage

Function: function swapFeeds(address creator, SlotSwapRequest[] call-data swaps) external

swapFeeds is a reorganization function by which a creator or admin swaps the positions and data of two feeds belonging to the same creator.

Inputs

- creator
 - **Control:** Creator/ADMIN_ROLE Only
 - **Constraints:** The caller must be the creator or an admin, and the namespace must be initialized.
 - **Impact:** Restricts the ownership scope of move/swap operations.
- swaps
 - **Control:** Creator/ADMIN_ROLE Only
 - **Constraints:** Both feeds must exist and be owned by the same creator.
 - **Impact:** Exchanges the positions and stored data of the two feeds.

Branches and code coverage

Intended branches

- If `msg.sender == creator`, the caller can process the swaps array within that creator namespace.

☒ Test coverage
- If `msg.sender` holds `ADMIN_ROLE`, it can perform the same swap processing on behalf of creator.

☒ Test coverage
- For each swap where both `feedIdA` and `feedIdB` exist and are owned by creator, the function reads the packed oracle data from the current (`slotIdx`, `pos`) positions of both feeds, exchanges their data, and updates `oracleInfo[feedIdA].misc` and `oracleInfo[feedIdB].misc` to point to each other's position.

☒ Test coverage

- A swap exchanges only the packed 48-bit oracle data and `oracleInfo[...].misc` pointers of the two positions, without updating the `_slotOccupancy` bitmap. Because both positions must be occupied before the swap (enforced by `PositionEmpty` checks), the occupancy bits remain unchanged after the swap.

☒ Test coverage

- If `swaps.length == 0`, the function passes only the permission/namespace checks and succeeds without modifying any feed data.

☐ Test coverage

- If multiple swaps are provided, they are executed sequentially starting from the first index; subsequent swaps operate on the updated `oracleInfo[feedIdA].misc` and `oracleInfo[feedIdB].misc` pointers left by earlier swaps.

☒ Test coverage

Negative branches

- If the caller is neither `creator` nor holds `ADMIN_ROLE`, the function reverts with `NotCreatorOrOwner()` and no swaps are processed.

☒ Test coverage

- If `namespaceRemapping[creator] != creator`, the function reverts with `InvalidNamespace()` before entering the array loop.

☐ Test coverage

- If `feedIdA` is not registered at any index, the function reverts with `FeedNotFound(feedIdA)`, rolling back the entire batch.

☐ Test coverage

- If `feedIdB` is not registered at any index, the function reverts with `FeedNotFound(feedIdB)`, rolling back the entire batch.

☐ Test coverage

- If the packed creator of `feedIdA` differs from the `creator` argument, the function reverts with `FeedNotOwnedByCreator(feedIdA)`, preventing feeds from different namespaces from being mixed.

☐ Test coverage

- If the packed creator of `feedIdB` differs from the `creator` argument, the function reverts with `FeedNotOwnedByCreator(feedIdB)`, preventing feeds from different namespaces from being mixed.

☒ Test coverage

- If the occupancy bit for the position pointed to by `feedIdA` is empty, the function reverts with `PositionEmpty(slotIdxA, posA)`, preventing a swap of empty slot data.

☐ Test coverage

- If the occupancy bit for the position pointed to by `feedIdB` is empty, the function reverts

with `PositionEmpty(slotIdxB, posB)`, preventing a swap of empty slot data.

- ☐ Test coverage
- Feed moves/swaps remap feed pointers but do not invalidate outstanding signed slot words. Unless existing signatures are drained/expired or signed against a layout nonce, old-position data may be applied to the new feed mapping.
- ☐ Test coverage

Function: `fallback()` override external

`fallback` is the calldata update path by which a creator self-namespace or delegated pusher directly pushes packed slot words to the creator namespace. The calldata format is `[deadline:4][slotWord:32]*`, and `msg.sender` must be mapped to a creator via `namespaceRemapping[msg.sender]`.

Inputs

- `msg.sender`
 - **Control:** Arbitrary
 - **Constraints:** `namespaceRemapping[msg.sender] != address(0)` must hold.
 - **Impact:** Determines which creator namespace slot is updated.
- `deadline(calldata[0:4])`
 - **Control:** Arbitrary
 - **Constraints:** `block.timestamp <= deadline` must hold; equality is permitted.
 - **Impact:** Prevents expired delegated pusher updates.
- `slotWord[] (calldata[4:calldatasize], 32-byte chunks)`
 - **Control:** Arbitrary
 - **Constraints:** The calldata length after the 4-byte deadline must consist of exactly aligned 32-byte words; each word must pass the low 8-bit slot ID, timestamp, and initialized-position data validity checks.
 - **Impact:** Updates the creator's compressed oracle slot storage.

Branches and code coverage

Intended branches

- If the caller is mapped as a creator self-namespace or delegated pusher and the deadline and calldata length are valid, the function iterates over each 32-byte word.

☒ Test coverage

- If the word's timestamp is newer than the existing slot timestamp, the function validates via `_isDataValid`, masks the low 8-bit slot ID, and writes the word to storage.

☒ Test coverage

- If the word's timestamp is not newer than the existing slot timestamp, the function skips that word and continues processing the next one.

☒ Test coverage

Negative branches

- If `namespaceRemapping[msg.sender] == address(0)`, the function reverts with `InvalidNamespace()`.

☒ Test coverage

- If the deadline has expired, the function reverts at the deadline check.

☒ Test coverage

- If the calldata length contains only the deadline or `(calldatasize - 4) % 32 != 0`, the function reverts with `BadCalldataLength()`.

☒ Test coverage

- If a word's `slotId > r.currentSlotIndex`, the function reverts with `InvalidSlotIndex()`.

☐ Test coverage

- If a word's timestamp exceeds the permitted future drift, the function reverts.

☐ Test coverage

- If `_isDataValid(r, slotId, word)` returns false, the data push is outside the range of initialized feed positions and the function reverts with `UnializedPush()`.

☐ Test coverage

- The fallback can be used not only by delegated pushers but also from the creator self-namespace. If the creator key is compromised, raw packed slots can be written directly.

☐ Test coverage

- The default registry value permits a pre-initialization timestamp-only write to slot 0, which can affect freshness ordering after feeds are subsequently created.

☐ Test coverage

Function: `function price(bytes32 feedId) external override view returns (uint256 mid, uint256 spread, uint16 volatility, uint256 refTime)`

`price` is the public on-chain read of `CompressedOracle` that directly returns the result of `_price(feedId)`. Unlike `providers/OracleBase` whose `price` is blocked by `ReadDisabled`, `CompressedOracle` **opens** this single-argument `price(feedId)` for anyone to call without a blacklist or pool-binding gate. `_price` only checks registration via the `feedExists(feedId)` modifier, then restores price/spread/volatility/timestamp from the compressed slot via `_getOracleData` and returns them. No post-processing (staleness, price guard, or step adjustment) is applied; the raw restored values are returned as-is.

Inputs

- `feedId`
 - **Control:** Arbitrary
 - **Constraints:** Must be a registered feed as checked by the `feedExists(feedId)` modifier in `_price` (otherwise `FeedNotFound`). There are no caller restrictions beyond this.
 - **Impact:** Determines which feed to read. The return value is the result of restoring the compressed slot.

Branches and code coverage

Intended branches

- If the feed is an active registered feed, the function returns the (`mid`, `spread`, `volatility`, `refTime`) restored from the slot.
 - ☒ Test coverage
- If the feed is deactivated (`creator == address(0)`), `_getOracleData` returns `spread = BPS_BASE`, `volatility = BPS_BASE`, `price = 0`, and `timestampMs = 0` (i.e., `mid 0`, `spread 10000`, `refTime 0`).
 - ☒ Test coverage
- If the compressed value at the slot position has `v == 0xff && s == 0xff` (empty slot marker), the function returns `BPS_BASE` for both `spread` and `volatility`.
 - ☒ Test coverage

Negative branches

- If the feed is not registered, the `feedExists` modifier in `_price` reverts with `FeedNotFound(feedId)`.
 - ☐ Test coverage
- If codebook decoding fails (`Codebook256.decode` is called with a non-existent index),

the function reverts with `CodebookDecodeFailed(index)`.

☐ Test coverage

- If the position index is outside the range 0–3, `_selectCompressedData` reverts with `InvalidPosition(index)`.

☐ Test coverage

- **Even a deactivated feed does not cause price to revert;** instead it returns degenerate values (price 0, spread `BPS_BASE`, `refTime` 0). Consumers must therefore explicitly detect sentinels such as `spread == BPS_BASE`, `price == 0`, or `refTime == 0`, or they may mistake an inactive feed for valid data (pre-confirmed observation).

☒ Test coverage

- Because this function is an open read that bypasses blacklist and pool inSwap binding, the abuse-protection of the providers family does not apply to `CompressedOracle`. Raw prices are exposed to anyone.

☐ Test coverage

Function: `beforeSwap(address sender, address, bool, int128, uint128, uint256, uint128 bidPriceX64, uint128 askPriceX64, bytes calldata)` external override onlyPool returns (bytes4)

`beforeSwap` is the hook callback invoked by the pool immediately before executing a swap, enforcing two subhook logic steps in sequence. First, `_beforeSwapAllowlist(msg.sender, sender)` checks whether `sender` is on the swap allowlist for the given pool (reverting with `NotAllowedToSwap` if not), then `_beforeSwapPriceVelocity(msg.sender, bidPriceX64, askPriceX64)` verifies that the mid-price rate of change across blocks does not exceed the configured ceiling. The `onlyPool` modifier validates that `msg.sender` is a pool deployed by the same `FACTORY` (i.e., `getImmutables().factory == FACTORY`). If all checks pass, the function returns `IMetricOmmHooks.beforeSwap.selector`.

Inputs

- `sender`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Passed by the pool as the swap initiator. Used as the allowlist key.
 - **Impact:** If `allowedSwapper[pool][sender]` is false, the swap is blocked with `NotAllowedToSwap`.
- `bidPriceX64 / askPriceX64`
 - **Control:** Arbitrary, but constrained by pool binding

- **Constraints:** Prices retrieved from the oracle and passed by the pool. Used $asMidPrice = (bid + ask) / 2$.
 - **Impact:** If the rate of change relative to the previously recorded mid exceeds the velocity ceiling, the function reverts with `PriceVelocityExceeded`.
- Remaining positional arguments (recipient, zeroForOne, amountSpecified, priceLimitX64, packedSlot0, hookData)
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Received without names and not used in this callback.
 - **Impact:** None.

Branches and code coverage

Intended branches

- When sender is on the allowlist and within the velocity ceiling, the function returns the normal selector.
 - ☒ Test coverage
- When `prevMid == 0` in the velocity state (first call), the function records only the current mid price and passes without checking the rate of change.
 - ☒ Test coverage
- When `maxChangePerBlockE18 == 0` (not configured), the velocity check is skipped and the function passes.
 - ☒ Test coverage

Negative branches

- If `getImmutables().factory != FACTORY` for `msg.sender`, the function reverts with `OnlyPool(msg.sender, FACTORY)` (via `onlyPool`).
 - ☐ Test coverage
- If `allowedSwapper[pool][sender] == false`, the function reverts with `NotAllowedToSwap`.
 - ☒ Test coverage
- If `actualSq > allowedSq` (the squared rate of change exceeds $maxChange^2 * (1 + blockDiff)$), the function reverts with `PriceVelocityExceeded(actualSq, allowedSq)`.
 - ☐ Test coverage
- The velocity check first **unconditionally updates** `lastMidPriceX64/lastUpdateBlock` to the current values before comparing with `prevMid/prevBlock`, so any state changes are invalidated by the transaction rollback if a revert occurs. The allowlist check is performed before the price check, so an unauthorized swapper never reaches the velocity update.

- ☐ Test coverage
- The comparison is performed using squares to avoid on-chain `sqrt`, and the permitted deviation is proportional to `sqrt(1 + blockDiff)`, relaxing the bound for larger block intervals.
- ☐ Test coverage

Function call analysis

- `IMetricOmmPool(msg.sender).getImmutables()` (inside `onlyPool`)
 - **What is controllable?** Only `msg.sender` is determinable; a non-standard pool is blocked by the factory mismatch.
 - **If the return value is controllable, how is it used and how can it go wrong?** The factory field must match `FACTORY` to pass. A forged pool is unlikely to pass this check.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If this view call reverts, the callback fails and the swap reverts. Since a single hook instance serves multiple pools from the same factory, per-pool state is isolated via mappings keyed by pool address.

Function: `beforeAddLiquidity(address, address owner, uint80, LiquidityDelta calldata, bytes calldata)` external view override `onlyPool` returns (bytes4)

`beforeAddLiquidity` is the view hook callback invoked by the pool immediately before adding liquidity, enforcing the deposit allowlist. `_beforeAddLiquidityAllowlist(msg.sender, owner)` checks whether the position owner is on the deposit allowlist for the given pool (reverting with `NotAllowedToDeposit` if not), and `onlyPool` validates that the caller is a pool from the same `FACTORY`. If all checks pass, the function returns `IMetricOmmHooks.beforeAddLiquidity.selector`.

Inputs

- `owner`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** The position owner passed by the pool. Used as the allowlist key.
 - **Impact:** If `allowedDepositor[pool][owner]` is false, the addition is blocked with `NotAllowedToDeposit`.
- Remaining positional arguments (`sender`, `salt`, `deltas`, `hookData`)

- **Control:** Arbitrary, but constrained by pool binding
- **Constraints:** Not used in this callback.
- **Impact:** None.

Branches and code coverage

Intended branches

- When owner is on the deposit allowlist, the function returns the normal selector.

☒ Test coverage

Negative branches

- If `getImmutables().factory != FACTORY` for `msg.sender`, the function reverts with `OnlyPool(msg.sender, FACTORY)`.

☐ Test coverage

- If `allowedDepositor[pool][owner] == false`, the function reverts with `NotAllowedToDeposit`.

☒ Test coverage

- Since the allowlist key is owner, even if a third party funds the position via the adder, the check is performed against the position owner. Accordingly, a third party can contribute funds to a position owned by an allowed owner.

☐ Test coverage

Function call analysis

- `IMetricOmmPool(msg.sender).getImmutables()` (inside `onlyPool`)
 - **What is controllable?** Only `msg.sender` is determinable; factory mismatch blocks unauthorized callers.
 - **If the return value is controllable, how is it used and how can it go wrong?** Only pools that pass the `factory` field validation can proceed.
 - **What happens if it reverts, reenters, or does other unusual control flow?** This is a `view` callback that does not modify external state. A revert causes the liquidity addition to fail.

Function: `afterSwap(address sender, address recipient, bool zeroForOne, int128 amountSpecified, uint128 priceLimitX64, uint256 packedSlot0Initial, uint256 packedSlot0Final, uint128 bidPriceX64, uint128 askPriceX64, int128 amount0Delta, int128 amount1Delta, uint256, bytes calldata) external override onlyPool returns (bytes4)`

`afterSwap` is the hook callback invoked by the pool after completing a swap (following balance validation), performing two subhook logic steps. `_afterSwapReport(...)` reports the swap result to the pool's active `priceProvider` on a best-effort basis (failures are silently ignored via `try/catch`). `_afterSwapOracleStopLoss(...)` computes the per-share value (two metrics: `token0` and `token1`) at the oracle mid price for the range of bins the swap traversed, and reverts the swap if either metric falls below the high watermark by more than `maxDrawdownE6`. `onlyPool` validates that the caller is a pool from the same `FACTORY`.

Inputs

- `packedSlot0Initial / packedSlot0Final`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** The pool packs and passes the `slot0` state before and after the swap. `curBinIdx` is unpacked to determine the bin range `[lo, hi]` to inspect.
 - **Impact:** Determines the set of bins examined by the stop-loss check.
- `bidPriceX64 / askPriceX64`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Oracle prices passed by the pool. Used as `midPriceX64 = (bid + ask) / 2`.
 - **Impact:** Used to compute per-share value metrics (`metricT0/metricT1`).
- `sender / recipient / zeroForOne / amountSpecified / priceLimitX64 / amount0Delta / amount1Delta`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Swap result metadata passed to the `priceProvider` report.
 - **Impact:** Composes the report payload but is not directly used in stop-loss evaluation.
- 13th positional argument (`protocolFeeAmount`) and `hookData`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Not used in this callback.
 - **Impact:** None.

Branches and code coverage

Intended branches

- When `oracleStopLossDrawdownE6[pool] == 0` (not configured), the stop-loss check is skipped and only the report is performed.
☒ Test coverage
- When `totalShares == 0` for a bin within the range, that bin is skipped.
☐ Test coverage
- When a metric is at or above the watermark, the watermark is updated to reflect the increase and the function passes.
☐ Test coverage
- When a metric is below the watermark but at or above the threshold, the watermark is retained and the function passes.
☐ Test coverage
- When the `priceProvider` report succeeds or fails (via `try/catch`), execution continues regardless.
☒ Test coverage

Negative branches

- If `getImmutables().factory != FACTORY` for `msg.sender`, the function reverts with `OnlyPool`.
☐ Test coverage
- If any bin's metric falls below `hwm * (1e6 - drawdown) / 1e6`, the function reverts with `OracleStopLossTriggered(binIdx, isToken0, metric, threshold)`.
☐ Test coverage
- The stop-loss tracks `token0` and `token1` metrics independently, and reverts if **either** falls below the threshold. The watermark is updated immediately upon increase and only moves upward.
☐ Test coverage
- The report is best-effort, so even if the `priceProvider` reverts or does not implement `reportSwap`, the swap is not affected. In contrast, a stop-loss revert occurs after balance validation, rolling back the entire swap.
☐ Test coverage

Function call analysis

- `IPriceProviderSwapReporter(priceProvider).reportSwap(...)`
`(_afterSwapReport)`

- **What is controllable?** The priceProvider is an address derived from the pool immutable or slot3. The report payload is derived from the swap result.
- **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used (the try block is empty).
- **What happens if it reverts, reenters, or does other unusual control flow?** All reverts are swallowed by catch (bytes memory) {}. A malicious or unimplemented priceProvider cannot block the swap, but if reportSwap attempts reentrancy, the pool's nonReentrant guard activates.
- PoolStateLibrary._multipleBinStates / _multipleBinTotalShares (_afterSwapOracleStopLoss)
 - **What is controllable?** The target bin range is determined by the curBinIdx in the before/after slot0 and is difficult for the caller to directly manipulate.
 - **If the return value is controllable, how is it used and how can it go wrong?** Bin state/shares are used to compute per-share value. The query is based on extsload, depending on the pool's storage layout.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A query failure causes the callback to revert, rolling back the swap. No external funds are moved.

Function: addLiquidity(address owner, uint80 salt, LiquidityDelta call-data deltas, bytes calldata callbackData, bytes calldata hookData)

MetricOmmPool.addLiquidity applies nonReentrant(PoolActions.ADD_LIQUIDITY) — an action-scoped transient reentrancy guard that stores the keccak256 action ID of "addLiquidity" while the function is executing, blocking any re-entry into any other guarded function on this pool — and then delegates to LiquidityLib.addLiquidity via DELEGATECALL. Within the pool's storage context, it completes array length validation and a beforeAddLiquidity hook call, then iterates over deltas.binIdxs. For each bin, it computes the position key as keccak256(abi.encode(owner, salt, int8(binIdx))), checks the minimum liquidity threshold, and then calculates amount0/1Scaled — using a fixed ratio based on initialScaledToken0/1PerShareE18 if no existing liquidity is present (binTotalSharesVal == 0) (token1 only if binIdx < curBinIdx, token0 only if binIdx > curBinIdx, or a curPosInBin-proportional mix if equal), or proportional to the current bin balance with ceiling rounding if existing liquidity is present — and immediately updates binState, binTotalShares, and positionBinShares. After the loop, it aggregates binTotals, converts to external units via _deltasScaledToExternal(..., Ceil), takes a balance0/1Before snapshot, calls metricOmmModifyLiquidityCallback to pull tokens, verifies the balance increase, calls the afterAddLiquidity hook, emits a LiquidityAdded event, and returns.

Inputs

- owner

- **Control:** Arbitrary
 - **Constraints:** None — any address including `address(0)` is allowed.
 - **Impact:** Used as the owner component in `_positionBinKey(owner, salt, binIdx)`, attributing position shares to the specified address. An attacker can add liquidity under another user's owner address, thereby increasing that address's position on their behalf (no funds are lost since `msg.sender` pays the tokens). If hooks are enabled, the arbitrary owner is also passed to `beforeAddLiquidity` / `afterAddLiquidity`.
- `salt`
 - **Control:** Arbitrary
 - **Constraints:** None.
 - **Impact:** The second component of `_positionBinKey`. The same owner can create independent positions by using different `salt` values. Collision attacks are prevented by keccak256 preimage resistance.
- `deltas`
 - **Control:** Arbitrary
 - **Constraints:** `binIdxs.length == shares.length` required; each `binIdx` must be within `[LOWEST_BIN, HIGHEST_BIN]`; `newUserShares >= minimalMintableLiquidity` must be satisfied.
 - **Impact:** Determines the list of bins to supply and the share amount for each bin. If `shares[i] == 0`, that bin is skipped. Very large `shares` values will overflow inside `_checkedMul`, causing a **revert** (since `_checkedMul` is a separate function, Solidity 0.8 automatic checks apply — it does not silently wrap).
- `callbackData`
 - **Control:** Arbitrary
 - **Constraints:** None.
 - **Impact:** Passed as-is to `metricOmmModifyLiquidityCallback` only. If the callback implementation unconditionally trusts this data (e.g., uses it to determine which token to transfer), it can be exploited by an attacker who crafts a malicious payload.
- `hookData`
 - **Control:** Arbitrary
 - **Constraints:** None.
 - **Impact:** Passed as-is to `beforeAddLiquidity` and `afterAddLiquidity` hooks only. Not forwarded to the ERC-20 callback. If a hook unconditionally trusts this data, it can be exploited.

Branches and code coverage

Intended branches

- If `deltas.binIdxs.length` is zero, the function immediately returns `(0, 0)`.
 - ☑ Test coverage
- If the `beforeAddLiquidity` hook is disabled (`HOOKS == address(0)` or `BEFORE_ADD_LIQUIDITY_FLAG` is not set), the function proceeds without calling the hook.
 - ☑ Test coverage
- If the `beforeAddLiquidity` hook is enabled, the function calls the hook contract and then proceeds normally.
 - ☑ Test coverage
- If `shares[i]` is zero, that bin is skipped via `continue`.
 - ☑ Test coverage
- If `binTotalShares[binIdx]` is zero and `binIdx < curBinIdx`, the bin receives its initial liquidity using only `token1` (`amount0Scaled = 0, amount1Scaled = ceilDiv(initialScaledToken1PerShareE18 * sharesToAdd, 1e18)`).
 - ☑ Test coverage
- If `binTotalShares[binIdx]` is zero and `binIdx > curBinIdx`, the bin receives its initial liquidity using only `token0` (`amount1Scaled = 0, amount0Scaled = ceilDiv(initialScaledToken0PerShareE18 * sharesToAdd, 1e18)`).
 - ☑ Test coverage
- If `binTotalShares[binIdx]` is zero and `binIdx == curBinIdx`, the bin receives its initial liquidity as a mixed `token0` and `token1` amount proportional to `curPosInBin`. Specifically:
 - `token0Proportion = type(uint104).max - curPosInBin`
 - `token1Proportion = curPosInBin`
 - `amount0Scaled = mulDiv(token0Proportion * initialScaledToken0PerShareE18, sharesToAdd, type(uint104).max * 1e18, Ceil)`
 - `amount1Scaled = mulDiv(token1Proportion * initialScaledToken1PerShareE18, sharesToAdd, type(uint104).max * 1e18, Ceil)`
 - That is, when `curPosInBin` is close to 0 (price near the bottom of the bin), the `token0` proportion is high and the `token1` proportion is low. When `curPosInBin` is close to `type(uint104).max` (price near the top of the bin), the `token1` proportion is high and the `token0` proportion is low. When `curPosInBin = 0`, `token1` is 0 and only `token0` is required; when `curPosInBin = type(uint104).max`, `token0` is 0 and only `token1` is required — these boundary values connect continuously with the `binIdx > curBinIdx` and `binIdx < curBinIdx` cases respectively.
 - [x] Test coverage
- If `binTotalShares[binIdx]` is greater than zero, the required amounts are calculated

proportionally to the existing balance using ceiling rounding (`ceilDiv(balance * sharesToAdd, totalShares)`).

☒ Test coverage

- If both `amount0Added` and `amount1Added` are greater than zero, both tokens must be transferred via the callback.

☒ Test coverage

- If both `amount0Added` and `amount1Added` are zero (i.e., all bins have `shares == 0`), the function proceeds without calling the callback.

☒ Test coverage

- If the `afterAddLiquidity` hook is enabled, the hook contract is called in the final step.

☐ Test coverage

- When liquidity is supplied to multiple bins simultaneously, the function iterates over all bins in a single loop.

☒ Test coverage

- If `newUserShares` exactly equals `minimalMintableLiquidity`, the minimum liquidity check passes.

☒ Test coverage

Negative branches

- If `deltas.binIdxs.length` does not equal `deltas.shares.length`, the function reverts with `LiquidityDeltaLengthMismatch`.

☐ Test coverage

- If any `binIdx` is below `lowestBin` or above `highestBin`, the function reverts with `InvalidBinIndex`.

☒ Test coverage

- If the cumulative shares after addition fall below `minimalMintableLiquidity`, the function reverts with `MinimalLiquidity`.

☒ Test coverage

- If balance validation fails after the callback (`balance0Before + amount0Added > balanceOf`), the function reverts with `InsufficientTokenBalance`.

☒ Test coverage

- If `_checkedMul` overflows due to a large `shares` value, Solidity 0.8 automatic overflow checks cause a revert independently of the call site's unchecked context, turning the overflow into a DOS vector for that bin.

☒ Test coverage

- If the `beforeAddLiquidity` hook reverts, the entire transaction reverts.

☒ Test coverage

- If the `afterAddLiquidity` hook reverts, the entire transaction reverts even after state changes and the callback have completed, rolling back all state.
 - ☐ Test coverage
- If `metricOmmModifyLiquidityCallback` reverts (e.g., due to a token transfer failure), the entire transaction reverts after state changes, rolling back all state.
 - ☐ Test coverage
- If `binState.token0BalanceScaled` or `token1BalanceScaled` exceeds the `uint104` maximum, `toUint104()` triggers a `SafeCast` revert.
 - ☐ Test coverage

Function call analysis

- `MetricHooks.beforeAddLiquidity(...)`
 - **What is controllable?** `msg.sender`, `owner`, `salt`, `deltas`, and `hookData` are passed to the hook contract. `callbackData` is not forwarded to the hook. Since `owner` is an arbitrary address, any access-control logic in the hook that relies on `owner` can be bypassed.
 - **If the return value is controllable, how is it used and how can it go wrong?** Only the selector match is verified. If the hook returns the correct selector, execution continues. A malicious hook could read state to gain front-running information or manipulate external state.
 - **What happens if it reverts, reenters, or does other unusual control flow?** On revert, the entire transaction reverts with `HookCallFailed` or the hook's own error. If the hook attempts reentrancy that writes to pool state, it is blocked by `nonReentrant`. However, side effects such as calling other contracts or manipulating an oracle cannot be prevented.
- `IERC20(ctx.token0).balanceOf(address(this))` (before snapshot)
 - **What is controllable?** The result of the ERC-20 `balanceOf` call is fundamentally controlled by the external token contract. For fee-on-transfer or rebasing tokens, the actual balance may differ from expectations.
 - **If the return value is controllable, how is it used and how can it go wrong?** `balance0Before` / `balance1Before` serve as the baseline for verifying the balance increase after the callback. If an attacker artificially inflates the pre-callback balance via a flash loan, the validation could be somewhat relaxed; however, reentrancy to artificially alter the prior balance on the same pool is blocked by `nonReentrant`.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If a malicious ERC-20 reverts on `balanceOf`, the entire transaction reverts. This would not occur under normal circumstances.
- `IMetricOmmModifyLiquidityCallback(msg.sender).metricOmmModifyLiquidityCallback(...)`

- **What is controllable?** `msg.sender` may be an arbitrary contract; `amount0Added`, `amount1Added`, and `callbackData` are passed. `hookData` is not forwarded here. **CEI pattern violation:** at this point, `binState`, `binTotalShares`, `positionBinShares`, and `binTotals` have all already been updated. In other words, during the callback, the pool's internal state has already changed before the token deposit.
 - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used. The purpose of the callback is to transfer tokens; compliance is verified afterward via balance checks.
 - **What happens if it reverts, reenters, or does other unusual control flow?** On revert, the entire transaction is rolled back (including state changes). **Reentrancy threat:** reentrancy into the same pool is blocked by `nonReentrant` (transient storage-based), but calling `addLiquidity` / `swap` on a **different pool** during the callback is permitted. This is a potential vector for cross-pool reentrancy attacks. Scenarios include: the callback executing a swap on another pool to manipulate the oracle price, or the callback removing liquidity from another pool and moving tokens so that the current pool's balance validation passes. Additionally, if `msg.sender` is not an EOA and has logic that executes encoded instructions from `data`, arbitrary code execution becomes possible.
- `MetricHooks.afterAddLiquidity(...)`
 - **What is controllable?** `msg.sender`, `owner`, `salt`, `deltas`, `amount0Added`, `amount1Added`, and `hookData` are passed to the hook. `callbackData` is not forwarded here. At this point, all state updates and token receipts have been completed.
 - **If the return value is controllable, how is it used and how can it go wrong?** Only the selector match is verified. If the hook returns the correct selector, execution completes. A poorly implemented or malicious hook could produce side effects, but the pool state is already finalized at this point.
 - **What happens if it reverts, reenters, or does other unusual control flow?** On revert, all state changes (including token receipt) are rolled back. This means the hook can unilaterally undo a successful liquidity addition, making the hook trust model critical. Even if the hook attempts to call another pool, `nonReentrant` only protects the current pool, so cross-pool calls are possible.

Function: `removeLiquidity(address owner, uint80 salt, LiquidityDelta calldata deltas, bytes calldata hookData)`

`MetricOmmPool.removeLiquidity` applies `nonReentrant(PoolActions.REMOVE_LIQUIDITY)` — an action-scoped transient reentrancy guard — and then delegates to `LiquidityLib.removeLiquidity` via `DELEGATECALL`. Unlike `addLiquidity`, it immediately reverts with `NotPositionOwner` if `msg.sender != owner`, categorically preventing proxy

withdrawals. It then calls the `beforeRemoveLiquidity` hook (forwarding `hookData`) and iterates over `deltas.binIdxs`. For each bin, it computes the position key via `_positionBinKey(owner, salt, int8(binIdx))`, checks the held shares, reverts with `MinimalLiquidity` if `newUserShares` falls in the range `(0, minimalMintableLiquidity)`, calculates the withdrawal amount using floor rounding as `_checkedMul(binState.tokenXBalanceScaled, sharesToRemove) / binTotalSharesVal`, and immediately updates `binState`, `binTotalShares`, and `positionBinShares`. After the loop, it subtracts from `binTotals`, converts to external units via `_deltasScaledToExternal(..., Floor)`, transfers tokens via `safeTransfer`, calls the `afterRemoveLiquidity` hook, emits a `LiquidityRemoved` event, and returns. State changes are completed before the token transfer (CEI-compliant), but the `afterRemoveLiquidity` hook is called after the token transfer — if the hook deliberately reverts, the entire transaction including the token receipt is rolled back.

Inputs

- `owner`
 - **Control:** Arbitrary
 - **Constraints:** Must equal `msg.sender`; any mismatch causes a `NotPositionOwner` revert.
 - **Impact:** Used as the token return destination address and in the position key (`_positionBinKey`) calculation; specifying the wrong owner causes the function itself to revert.
- `salt`
 - **Control:** Arbitrary
 - **Constraints:** Must match the salt used during `addLiquidity` for the position to be found; there is no explicit validation (wrong salt $\rightarrow$ balance 0 $\rightarrow$ `InsufficientLiquidity`).
 - **Impact:** Participates in the `_positionBinKey(owner, salt, binIdx)` calculation; the same owner can manage multiple positions using different salts.
- `deltas`
 - **Control:** Arbitrary
 - **Constraints:** `binIdxs.length == shares.length` required; each `binIdx` must be within `[LOWEST_BIN, HIGHEST_BIN]`; if `shares[i] > positionBinShares[posKey]`, reverts with `InsufficientLiquidity`; if the remaining balance after removal falls in the range `(0, minimalMintableLiquidity)`, reverts with `MinimalLiquidity`.
 - **Impact:** Determines the list of bins to remove from and the number of shares to burn for each bin. If `shares[i] == 0`, that bin is skipped. Very large shares values will overflow inside `_checkedMul`, causing a **revert** (DOS vector).
- `hookData`
 - **Control:** Arbitrary

- **Constraints:** None.
- **Impact:** Passed as-is to both `beforeRemoveLiquidity` and `afterRemoveLiquidity` hooks. No ERC-20 callback is involved in `removeLiquidity`, so this is exclusively for hooks.

Branches and code coverage

Intended branches

- If `deltas.binIdxs.length` is zero, the function returns `(0, 0)` early.
 - ☒ Test coverage
- If `shares[i]` is zero, that bin is skipped.
 - ☐ Test coverage
- If all shares are burned (`newUserShares == 0`), the position is fully closed and its state is reset.
 - ☒ Test coverage
- If some shares remain after burning and `newUserShares >= minimalMintableLiquidity`, the partial withdrawal succeeds.
 - ☒ Test coverage
- If the `beforeRemoveLiquidity` hook is enabled, the function calls the hook and then proceeds normally.
 - ☐ Test coverage
- If the `afterRemoveLiquidity` hook is enabled, the hook is called after the token transfer.
 - ☐ Test coverage
- When withdrawing a token with fewer than 18 decimals, a fractional amount may be lost due to floor rounding during the scaled-to-external conversion.
 - ☐ Test coverage

Negative branches

- If `msg.sender` does not equal `owner`, the function reverts with `NotPositionOwner`.
 - ☐ Test coverage
- If `deltas.binIdxs.length` does not equal `deltas.shares.length`, the function reverts with `LiquidityDeltaLengthMismatch`.
 - ☐ Test coverage
- If any `binIdx` is below `LOWEST_BIN` or above `HIGHEST_BIN`, the function reverts with `InvalidBinIndex`.
 - ☐ Test coverage

- If `userShares` is less than `sharesToRemove`, the function reverts with `InsufficientLiquidity(sharesToRemove, userShares)`.
 - ☒ Test coverage
- If `newUserShares` is positive but less than `minimalMintableLiquidity`, the function reverts with `MinimalLiquidity(newUserShares, minimalMintableLiquidity)`.
 - ☒ Test coverage
- If the `beforeRemoveLiquidity` hook reverts, the entire transaction reverts without any state changes.
 - ☐ Test coverage
- If the `afterRemoveLiquidity` hook reverts, the entire transaction reverts even after the token transfer has occurred.
 - ☐ Test coverage
- If `safeTransfer` fails (e.g., due to insufficient balance or a blacklisted token), the entire transaction reverts.
 - ☐ Test coverage
- If `_checkedMul` overflows (`binState.token0BalanceScaled * sharesToRemove > type(uint256).max`), the function reverts, creating a DOS vector.
 - ☐ Test coverage

Function call analysis

- `MetricHooks.beforeRemoveLiquidity(...)`
 - **What is controllable?** `msg.sender`, `owner`, `salt`, `deltas`, and `hookData` are passed to the hook contract. `hookData` is the caller-supplied parameter (no longer hardcoded to `new bytes(0)`).
 - **If the return value is controllable, how is it used and how can it go wrong?** Only the selector match is verified. A malicious hook could read state to gain front-running information or manipulate external state.
 - **What happens if it reverts, reenters, or does other unusual control flow?** On revert, the entire transaction reverts. At this point the pool state has not yet changed, so it is safe. If the hook attempts reentrancy that writes to pool state, it is blocked by `nonReentrant`. However, side effects such as calling other contracts or manipulating an oracle cannot be prevented.
- `IERC20(ctx.token0).safeTransfer(owner, amount0Removed)`
 - **What is controllable?** `owner` equals `msg.sender` (caller-controlled); `amount0Removed` is determined by internal calculation.
 - **If the return value is controllable, how is it used and how can it go wrong?** `safeTransfer` checks the bool return value (reverts on false); standard ERC-20 tokens are safe, but for ERC-777 or tokens with transfer callbacks, a

callback during `safeTransfer` → pool reentrancy is blocked by the `nonReentrant` guard; tokens with blacklist/freeze functionality (e.g., USDC) may revert if owner is blacklisted → the withdrawal for that position becomes permanently impossible (DOS).

- **What happens if it reverts, reenters, or does other unusual control flow?**

On revert, the entire transaction is cancelled (pool state rolled back); the already-decremented shares and balance are restored. **CEI-compliant:** state changes (shares, `binState`, `binTotals`) are completed before token transfer, so when the `afterRemoveLiquidity` hook is called after a successful transfer, the pool state already reflects the final state.

- `MetricHooks.afterRemoveLiquidity(...)`

- **What is controllable?** The hook address is immutable; sender (= `msg.sender`), owner, salt, deltas, `amount0Removed`, `amount1Removed`, and `hookData` are passed. `hookData` is the caller-supplied parameter (no longer hardcoded to `new bytes(0)`).

- **If the return value is controllable, how is it used and how can it go wrong?**

Only the return of the `IMetricOmmHooks.afterRemoveLiquidity` selector is verified. At this point, **tokens have already been transferred to owner** and pool state (shares, `binState`, `binTotals`) has already been updated. If the hook reverts, the entire transaction is rolled back including the transferred tokens → an untrusted hook contract could selectively block withdrawals; a malicious hook may attempt pool reentrancy, but `nonReentrant` blocks all `nonReentrant` functions on the same pool; side effects from the hook calling other external contracts or performing oracle manipulation are not protected by `nonReentrant`.

- **What happens if it reverts, reenters, or does other unusual control flow?**

Hook revert → entire transaction cancelled; this maintains consistency even though tokens have already been transferred; however, if the hook deliberately performs selective reverts, it can permanently block specific LPs from removing liquidity (griefing) → the trustworthiness of the hook contract is critical.

Function: `swap(address recipient, bool zeroForOne, int128 amountSpecified, uint128 priceLimitX64, bytes callData callData, bytes callData hookData)`

`swap` is the core trading function of `MetricOmmPool`. It applies when `NotPaused` and `nonReentrant(PoolActions.SWAP)` — an action-scoped transient reentrancy guard that stores the keccak256 action ID of "swap" for the duration of execution, blocking re-entry into any guarded function on this pool. Notably, the `inSwap()` view function returns the resolved price provider address whenever `_currentAction() == PoolActions.SWAP`, allowing external contracts (such as price providers) to detect they are being called during a live swap. The function fetches bid/ask prices from the CEX oracle (`IPriceProvider`) to derive a geometric-mean `midPriceX64`, then

executes a token exchange within bins at that price. The sign of `amountSpecified` determines the mode: positive means `exactIn` (exact input amount) and negative means `exactOut` (exact output amount). Combined with the `zeroForOne` direction flag (`token0→token1` or `token1→token0`), this produces four distinct execution paths. After the swap, the function first sends the output tokens to `recipient`, then invokes `IMetricOmmSwapCallback` on the caller to allow it to settle the input tokens — a flash-swap pattern (inverted CEI). Two fee mechanisms apply: a spread fee (derived from the oracle bid/ask spread) and a notional fee (proportional to trade size); the notional fee formula differs between `exactIn` and `exactOut` modes.

Inputs

- `recipient`
 - **Control:** Arbitrary
 - **Constraints:** No `address(0)` validation is performed.
 - **Impact:** If misconfigured, output tokens are sent to the wrong address with no recovery mechanism.
- `zeroForOne`
 - **Control:** Arbitrary
 - **Constraints:** `true` means paying `token0` and receiving `token1` (price decreases); `false` means the opposite direction.
 - **Impact:** Determines which of the four swap paths is taken and which token's balance is validated after the callback.
- `amountSpecified`
 - **Control:** Arbitrary
 - **Constraints:** `int128`; maximum absolute value is `type(int128).max`. Must be non-zero; a value of 0 triggers an `InvalidAmount` revert.
 - **Impact:** Positive selects `exactIn` mode (exact input amount); negative selects `exactOut` mode (exact output amount). Determines how `amountInScaled` or `amountOutScaled` is derived, including the scale-multiplier branching.
- `priceLimitX64`
 - **Control:** Arbitrary
 - **Constraints:** Expressed as a Q64.64 fixed-point price. Acts as a lower bound when `zeroForOne=true` (swap is halted if the current price is already below this limit); acts as an upper bound when `false`.
 - **Impact:** Setting this to 0 removes all price guardrails, allowing the swap to traverse all bins and exhaust the entire pool's liquidity.
- `callbackData`
 - **Control:** Arbitrary
 - **Constraints:** No validation or length limit.
 - **Impact:** Passed exclusively to

`IMetricOmmSwapCallback.metricOmmSwapCallback`. Allows the caller to encode router context, flash-loan repayment logic, or other callback-side state. Not forwarded to hooks.

- `hookData`
 - **Control:** Arbitrary
 - **Constraints:** No validation or length limit.
 - **Impact:** Passed exclusively to `beforeSwap` and `afterSwap` hooks. Not forwarded to the ERC-20 callback. If a hook unconditionally trusts this data, it can be exploited.

Branches and code coverage

Intended branches

- If `IMMUTABLE_PRICE_PROVIDER != address(0)`, `_resolvedPriceProvider()` uses the immutable oracle path.
 - ☒ Test coverage
- If `IMMUTABLE_PRICE_PROVIDER == address(0)`, `_resolvedPriceProvider()` uses the mutable `priceProvider` storage.
 - ☒ Test coverage
- If `HOOKS != address(0)` and the `BEFORE_SWAP_FLAG` or `AFTER_SWAP_FLAG` is set, the corresponding hook is called.
 - ☒ Test coverage
- If `HOOKS == address(0)`, all hooks are disabled and skipped.
 - ☒ Test coverage
- When `amountSpecified > 0` and `zeroForOne == true` (`exactIn`, `token0` → `token1`), the function swaps an exact `token0` input for `token1` output.
 - ☒ Test coverage
- When `amountSpecified > 0` and `zeroForOne == false` (`exactIn`, `token1` → `token0`), the function swaps an exact `token1` input for `token0` output.
 - ☒ Test coverage
- When `amountSpecified < 0` and `zeroForOne == true` (`exactOut`), the function delivers an exact `token1` output in exchange for `token0` payment.
 - ☒ Test coverage
- When `amountSpecified < 0` and `zeroForOne == false` (`exactOut`), the function delivers an exact `token0` output in exchange for `token1` payment.
 - ☒ Test coverage
- If `priceLimitX64` is zero, no price guardrail is enforced and the swap may traverse all bins, exhausting the entire pool's liquidity.

☒ Test coverage

- If `priceLimitX64` is already exceeded at the current price before the swap begins, `_swapToken...` returns (0, 0, 0) immediately, `amount0Delta` and `amount1Delta` are both zero, the callback is still invoked, but balance validation is skipped entirely.

☐ Test coverage

- If `priceLimitX64` is reached mid-swap, the while loop exits early and the full requested amount is not filled.

☒ Test coverage

- When the swap is completed within a single bin without crossing a bin boundary, only one bin's state is updated.

☒ Test coverage

- When a swap crosses a bin boundary, the function moves to the next bin and continues until the amount is fully filled or a limit is reached.

☒ Test coverage

- When a multi-bin swap encounters a bin with zero balance, that bin is skipped and the swap continues to the next bin.

☐ Test coverage

- If `amountOutScaled` exceeds `totalAvailableTokenScaled`, the requested output exceeds total bin liquidity and is automatically clamped.

☒ Test coverage

- When an `exactOut` swap requests an amount equal to the total pool liquidity, the pool is fully drained.

☐ Test coverage

- If `spreadFeeE6` is greater than zero, the spread fee is applied to the swap.

☒ Test coverage

- If `binState.addFeeBuyE6` or `addFeeSellE6` is greater than zero, the per-bin additional fee is applied.

☐ Test coverage

- If `notionalFeeE8` is greater than zero, the notional fee is applied and accumulated.

☒ Test coverage

- If `notionalFeeE8` is zero, no notional fee is applied.

☒ Test coverage

- If `amount0Delta > 0` (when `zeroForOne == true`), balance validation is executed to confirm the input token has been settled.

☒ Test coverage

- If `amount1Delta > 0` (when `zeroForOne == false`), balance validation is executed to confirm the input token has been settled.

☒ Test coverage

Negative branches

- If amountSpecified is zero, the function reverts with InvalidAmount.

☒ Test coverage

- If pauseLevel is non-zero, the whenNotPaused modifier causes the function to revert with PoolPaused.

☒ Test coverage

- If the price provider reverts during _getBidAndAskPriceX64(), the function reverts with PriceProviderFailed.

☐ Test coverage

- If the oracle returns a bid price greater than the ask price, the function reverts with BidGreaterThanAsk.

☐ Test coverage

- If the oracle returns a bid price of zero, the function reverts with BidIsZero.

☐ Test coverage

- If the input token balance after the callback is insufficient, the function reverts with IncorrectDelta.

☐ Test coverage

- If amountSpecifiedScaled exceeds type(uint128).max inside _getInitialStateForSwap, the function reverts with AmountScaledExceedsMax.

☐ Test coverage

- If the BEFORE_SWAP hook reverts, the error bubbles up as HookCallFailed or the hook's original error.

☒ Test coverage

- If the AFTER_SWAP hook reverts, the error bubbles up as HookCallFailed or the hook's original error.

☐ Test coverage

- If a hook returns an invalid selector, the function reverts with InvalidHookResponse.

☐ Test coverage

- If a reentrancy attempt is detected, the transient storage-based guard causes a revert with ReentrancyGuardReentrantCall.

☒ Test coverage

Function call analysis

- `_getBidAndAskPriceX64()` / `IPriceProvider(activePriceProvider).getBidAndAskPrice()`
 - **What is controllable?** `priceProvider` may be mutable (replaceable via `setPriceProvider` by a factory-privileged actor) or immutable depending on deployment. If mutable, the factory admin can point it to an arbitrary address. If the active `priceProvider` depends on an external CEX API feed, manipulation or errors in that feed are reflected directly.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned bid/ask prices flow into `_midAndSpreadFeeX64FromBidAsk`, which computes `midPriceX64` (geometric mean) and `baseFeeX64` (spread-based fee). Artificially inflating or deflating the oracle price shifts the actual exchange rate within bins, allowing an attacker to trade at a favorable price relative to LPs or to drive ordinary users into disadvantageous swaps.
 - **What happens if it reverts, reenters, or does other unusual control flow?** The call is wrapped in `try/catch`; all reverts are converted to `PriceProviderFailed(reason)`. If the `priceProvider` attempts reentrancy before the catch, this pool's `nonReentrant` guard blocks reentry into this specific pool, but the same `priceProvider` could still call other pools via a cross-pool reentrancy pattern.
- `MetricHooks.beforeSwap(...)`
 - **What is controllable?** `HOOKS` is immutable and cannot be changed after deployment; `HOOKS_PERMISSIONS` is likewise fixed. The `hookData` payload passed to the hook is fully controlled by the caller. `callbackData` is not forwarded to hooks. `bidPriceX64` and `askPriceX64` are passed directly from the live oracle result and `packedSlot0Initial` is derived from the pool; neither can be directly controlled by an attacker.
 - **If the return value is controllable, how is it used and how can it go wrong?** `callHook` verifies that the returned selector matches the called selector; a mismatch triggers `InvalidHookResponse`. The return value has no further influence on swap logic.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A hook revert bubbles up as the original error or `HookCallFailed`, failing the entire swap. If the hook maliciously attempts to reenter this pool while `nonReentrant` is held, the guard blocks it. However, the hook is free to call other contracts. Because `beforeSwap` executes before the swap itself, a malicious hook could read pool state to enable sandwich MEV attacks or conditionally block swaps.
- `transferToken0/1(recipient, ...)`
 - **What is controllable?** `recipient` is fully controlled by the caller. The transfer amount is the swap output derived from pool-internal calculations.
 - **If the return value is controllable, how is it used and how can it go wrong?** Uses `OpenZeppelin SafeERC20.safeTransfer`. The output token is sent to

recipient **before** the input token balance is validated — the canonical inverted-CEI / flash-swap pattern intentionally borrowed from Uniswap V2/V3. If recipient is a malicious contract it receives the output tokens before providing the input, but the subsequent `IncorrectDelta` check enforces settlement. If the token is an ERC-777 or an ERC-20 with hooks enabled, `safeTransfer` may trigger a recipient callback; within that callback the `nonReentrant(PoolActions.SWAP)` action-scoped guard blocks reentry into any guarded function on **this** pool, but calls to **other** pools or contracts are not restricted.

- **What happens if it reverts, reenters, or does other unusual control flow?** A transfer revert (e.g., insufficient pool balance, blocklisted recipient) rolls back the entire swap. Same-pool reentrancy during an ERC-20 callback is blocked by `nonReentrant`; cross-pool calls are not blocked.
- `IMetricOmmSwapCallback(msg.sender).metricOmmSwapCallback(amount0Delta, amount1Delta, callbackData)`
 - **What is controllable?** The caller provides the callback implementation via `msg.sender`, giving it full control over all callback logic. `amount0Delta`, `amount1Delta`, and `callbackData` are all passed through; `hookData` is not forwarded here. The callback implementation is expected to parse these and send the input tokens to the pool.
 - **If the return value is controllable, how is it used and how can it go wrong?** The pool neither uses nor validates the callback's return value. The caller's sole obligation is to ensure that the pool's token balance has increased by the required delta before the subsequent balance check. Since the output token has already been sent, an attacker who implements the callback but withholds the input tokens will be caught by the `IncorrectDelta` revert. The typical legitimate use case is a cross-pool flash-loan pattern: the callback takes the output tokens from this pool, uses them in another pool or protocol to obtain the input tokens, and returns the input tokens to this pool within the same transaction.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A callback revert causes a full transaction rollback, undoing all pool state changes, token transfers, and the transient `nonReentrant` lock. Same-pool reentrancy from inside the callback is blocked by `nonReentrant`. Calls to other pools or external contracts from inside the callback are not restricted.
- Balance validation (`balance0()` / `balance1()` check)
 - **What is controllable?** Not directly controllable; `balance0/1()` calls `IERC20.balanceOf(address(this))` on the immutable pool token addresses.
 - **If the return value is controllable, how is it used and how can it go wrong?** For `zeroForOne`: if `amount0Delta > 0` and `balance0Before + uint256(amount0Delta) > balance0()`, the function reverts with `IncorrectDelta`. The check compares the expected post-transfer balance against the actual balance obtained via `balanceOf`. Because `balanceOf` is used directly, fee-on-transfer tokens will always fail this check when the

actual amount received is less than `amount0Delta` due to fees. Additionally, if `amount0Delta == 0` the check is skipped entirely, meaning an unusual zero-input swap bypasses balance validation.

- **What happens if it reverts, reenters, or does other unusual control flow?**

An `IncorrectDelta` revert rolls back everything including the output token transfer. No external calls beyond the `balanceOf` query.

- `MetricHooks.afterSwap(...)`

- **What is controllable?** `hookData` is caller-controlled and forwarded to the hook. `callbackData` is not forwarded here. The hook contract itself is immutable. `packedSlot0Initial` and `packedSlot0Final` capture pool state before and after the swap and are derived from the pool; `bidPriceX64` and `askPriceX64` are passed directly from the oracle result obtained at the start of the swap.

- **If the return value is controllable, how is it used and how can it go wrong?** Called after balance validation has already succeeded, so the hook receives the verified `amount0Delta`, `amount1Delta`, and `protocolFeeAmount`. The hook may emit events, update external accounting, or perform additional logic based on the finalized swap outcome. Because it executes after all pool state has been committed, any side effects (including state changes in other contracts) cannot be undone without reverting the entire transaction.

- **What happens if it reverts, reenters, or does other unusual control flow?** A revert rolls back the entire transaction including the swap, output transfer, and balance validation. Same-pool reentrancy is blocked by `nonReentrant`. Cross-pool calls from `afterSwap` are not restricted, enabling complex composite attack patterns where the hook leverages the confirmed swap outcome to interact with other pools or protocols.

Function: `collectFees(uint256 protocolSpreadFeeE6_, uint256 adminSpreadFeeE6_, uint256 protocolNotionalFeeE8_, uint256 adminNotionalFeeE8_, address adminFeeDestination_)` external onlyFactory nonReentrant

`collectFees` is an `onlyFactory` function that distributes accumulated pool fees to two recipients: `admin` and `protocol`. Fees are divided into two categories: spread fees (based on the surplus pool balance after subtracting bin totals and notional fees) and notional fees (based on `notionalFeeToken0Scaled/notionalFeeToken1Scaled` accumulated during swaps). Each fee type is split between `admin` and `protocol` according to the `protocolSpreadFeeE6_` and `adminSpreadFeeE6_` ratios, using floor rounding so that dust amounts remain in the pool. At the end of the function, `notionalFeeToken0Scaled` and `notionalFeeToken1Scaled` are reset to zero.

Inputs

- `protocolSpreadFeeE6_`:
 - **Control:** Factory-supplied
 - **Constraints:** The function operates even if the sum with `adminSpreadFeeE6_` is 0 (spread fee distribution is skipped); `uint256` type but the actual value is managed as `uint24` in the factory.
 - **Impact:** Determines the protocol's share of the spread fee.
- `adminSpreadFeeE6_`:
 - **Control:** Factory-supplied
 - **Constraints:** Same as above.
 - **Impact:** Determines the admin's share of the spread fee; if the sum of both spread fees is 0, the entire spread fee distribution is skipped.
- `protocolNotionalFeeE8_`:
 - **Control:** Factory-supplied
 - **Constraints:** If the sum with `adminNotionalFeeE8_` is 0, notional fee distribution is skipped.
 - **Impact:** Determines the protocol's share of the notional fee.
- `adminNotionalFeeE8_`:
 - **Control:** Factory-supplied
 - **Constraints:** Same as above.
 - **Impact:** Determines the admin's share of the notional fee.
- `adminFeeDestination_`:
 - **Control:** Factory-supplied
 - **Constraints:** No on-chain validation against `address(0)` (validation is performed by the factory at pool creation).
 - **Impact:** The address that receives `token0` and `token1` admin fees; if this is a malicious contract, a `safeTransfer` failure can revert the entire function.

Branches and code coverage

Intended branches

- If both `spreadSumE6 > 0` and `notionalSumE8 > 0`, both fee types are distributed to the admin and protocol.
 - ☐ Test coverage
- If `spreadSumE6 == 0`, the spread fee distribution is skipped and only the notional fee is distributed.
 - ☒ Test coverage

- If `notionalSumE8 == 0`, the notional fee distribution is skipped and only the spread fee is distributed.

☒ Test coverage

- If both `spreadSumE6 == 0` and `notionalSumE8 == 0`, the function returns early to save gas.

☐ Test coverage

- If the surplus is zero, all fee amounts (`totalFee0ToAdmin`, `totalFee1ToAdmin`, etc.) are zero and transfers are skipped.

☐ Test coverage

- If `totalFee0ToAdmin` or `totalFee1ToAdmin` is zero, the corresponding transfer is conditionally skipped.

☒ Test coverage

Negative branches

- If `msg.sender` is not `FACTORY`, the function reverts with `OnlyFactory`.

☐ Test coverage

- If a reentrancy attempt is detected, the `nonReentrant` modifier causes a revert.

☐ Test coverage

- If `adminFeeDestination_` is a malicious contract that reverts on the `token0` transfer, the `token1` transfer is also not executed and the entire call rolls back, preventing partial distribution.

☐ Test coverage

- If the pool balance is less than `binTotals + notionalFee`, an underflow occurs in the unchecked block during the `surplus0Scaled` calculation; this is theoretically impossible under normal operation but may occur with fee-on-transfer or rebasing tokens.

☐ Test coverage

Function call analysis

- `transferToken0(adminFeeDestination_, totalFee0ToAdmin)`
 - **What is controllable?** `adminFeeDestination_` is an address set by the admin via the factory; the transfer amount is a computed value.
 - **If the return value is controllable, how is it used and how can it go wrong?** Uses `safeTransfer`, so any failure causes a revert; if the admin fee destination is a malicious contract, it could revert after receiving the `token0` transfer but before the `token1` transfer (reentrancy is prevented by `nonReentrant`).

- **What happens if it reverts, reenters, or does other unusual control flow?** A revert rolls back the entire `collectFees` call; reentrancy is prevented by `nonReentrant`; in theory, if the admin token0 transfer succeeds but the protocol token0 transfer fails, a partial distribution where admin receives token0 and protocol receives nothing could occur — however, this is also protected by a full revert.
- `transferToken1(FACTORY, totalFee1ToProtocol)`
 - **What is controllable?** The recipient address is the immutable `FACTORY`; the amount is a computed value.
 - **If the return value is controllable, how is it used and how can it go wrong?** The factory contract is the recipient and can be trusted.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Even if the factory has no `receive()` or fallback, the ERC-20 transfer succeeds without reverting; if the factory has been destroyed, the transfer still succeeds (effectively burning the tokens).

Function: `setPoolFees(uint24 newSpreadFeeE6, uint24 newNotionalFeeE8)` **external onlyFactory nonReentrant**

`setPoolFees` is an `onlyFactory` function that updates the pool's `spreadFeeE6` (spread fee) and `notionalFeeE8` (notional fee). The factory passes the combined sum of the protocol fee and admin fee, and the pool stores this combined value as a single state variable. State is updated and an event is emitted only when a change occurs. Both fees are packed state variables in Slot 0, so calling this function writes to Slot 0 and may conflict with the Slot 0 reads of concurrently submitted swap transactions (though EVM transaction serialization means there is no true concurrent execution).

Inputs

- `newSpreadFeeE6`:
 - **Control:** Factory-supplied
 - **Constraints:** `uint24` type (maximum 16,777,215); validated to be at or below `maxProtocolSpreadFeeE6 + maxAdminSpreadFeeE6` within the factory; no separate validation in the pool itself.
 - **Impact:** Used directly in spread fee calculations during swaps; a high value effectively acts as a near-total swap rejection.
- `newNotionalFeeE8`:
 - **Control:** Factory-supplied
 - **Constraints:** `uint24` type; validated against a cap in the factory; used in the denominator calculation `1e8 - notionalFeeE8` in the pool, so a value of

`notionalFeeE8 >= 1e8` would cause a division error (theoretically safe since `type(uint24).max < 1e8`).

- **Impact:** Used in the gross-up calculation for exactOut swaps; a high value significantly increases the cost of exactOut swaps.

Branches and code coverage

Intended branches

- If `newSpreadFeeE6` differs from the current `spreadFeeE6`, the state variable is updated and a `SpreadFeeUpdated` event is emitted.
 - ☒ Test coverage
- If `newNotionalFeeE8` differs from the current `notionalFeeE8`, the state variable is updated and a `NotionalFeeUpdated` event is emitted.
 - ☒ Test coverage
- If both `newSpreadFeeE6` and `newNotionalFeeE8` are unchanged, no state changes occur and no events are emitted.
 - ☐ Test coverage
- If both fee values change, both `SpreadFeeUpdated` and `NotionalFeeUpdated` events are emitted.
 - ☐ Test coverage

Negative branches

- If `msg.sender` is not `FACTORY`, the function reverts with `OnlyFactory`.
 - ☐ Test coverage
- If a reentrancy attempt is detected, the `nonReentrant` modifier causes a revert.
 - ☐ Test coverage

Function: `setPause(uint8 newLevel)` external onlyFactory

`setPause` is an `onlyFactory` function that sets the pool's `pauseLevel` to one of 0, 1, or 2. `pauseLevel 0` represents normal operation, 1 represents an admin-initiated pause, and 2 represents a protocol-initiated pause. The `whenNotPaused` modifier is applied only to the `swap` function; `addLiquidity` and `removeLiquidity` continue to execute even when the pool is paused. This is an intentional design choice to ensure that liquidity providers (LPs) can always withdraw their funds under any circumstances. The function has no `nonReentrant` modifier, but since it only modifies a simple state variable without making any external calls, there is no reentrancy risk.

Inputs

- `newLevel`:
 - **Control:** Factory-supplied
 - **Constraints:** Only 0, 1, and 2 are valid; a value greater than 2 triggers an `InvalidPauseLevel` revert; if `newLevel == pauseLevel`, an early return is executed.
 - **Impact:** 0 resumes swaps; 1 or 2 blocks swaps; no effect on `addLiquidity/removeLiquidity`.

Branches and code coverage

Intended branches

- If `newLevel` is 1 (admin pause), swap operations are blocked while liquidity addition and removal remain possible.
 - ☒ Test coverage
- If `newLevel` is 2 (protocol pause), swap operations are blocked while liquidity addition and removal remain possible.
 - ☒ Test coverage
- If `newLevel` is 0, the pool is unpaused and swap operations are resumed.
 - ☒ Test coverage
- If `newLevel` equals the current `pauseLevel`, the function returns early and no event is emitted.
 - ☐ Test coverage

Negative branches

- If `msg.sender` is not `FACTORY`, the function reverts with `OnlyFactory`.
 - ☐ Test coverage
- If `newLevel` exceeds 2, the function reverts with `InvalidPauseLevel`.
 - ☐ Test coverage

Function: `setBinAdditionalFees(int8 bin, uint16 addFeeBuyE6, uint16 addFeeSellE6)` external onlyFactory nonReentrant

`setBinAdditionalFees` is an `onlyFactory` function that sets the buy (`addFeeBuyE6`) and sell (`addFeeSellE6`) additional fees for a specific bin. These additional fees are added to `baseFeeX64` during swaps, increasing the effective fee for that bin. After validating the bin range, the function

directly updates the two fields in `_binStates[bin]`. `nonReentrant` is applied, though there are no external calls and therefore no actual reentrancy risk.

Inputs

- `bin`:
 - **Control**: Factory-supplied
 - **Constraints**: Must satisfy `LOWEST_BIN <= bin <= HIGHEST_BIN`; values outside this range trigger an `InvalidBinIndex(bin)` revert; `int8` type, range -128 to 127.
 - **Impact**: Determines which bin's additional fees are modified; changing the fees for the currently active bin (`curBinIdx`) takes effect immediately on the next swap.
- `addFeeBuyE6`:
 - **Control**: Factory-supplied
 - **Constraints**: `uint16` type (maximum 65,535 $\approx$ 6.55%); no upper bound validation in the pool itself (managed by the factory).
 - **Impact**: The additional fee applied when buying token0 (token1 $\rightarrow$ token0) in the given bin; combined as `baseFeeX64 + mulDiv(addFeeBuyE6, ONE_X64, 1e6)`.
- `addFeeSellE6`:
 - **Control**: Factory-supplied
 - **Constraints**: `uint16` type; no upper bound validation in the pool itself.
 - **Impact**: The additional fee applied when buying token1 (token0 $\rightarrow$ token1) in the given bin.

Branches and code coverage

Intended branches

- When called with a valid bin index, `_binStates[bin].addFeeBuyE6` and `addFeeSellE6` are updated and a `BinAdditionalFeesUpdated` event is emitted.
 - ☒ Test coverage
- If `bin` equals the currently active `curBinIdx`, the new fees take effect immediately on the next swap.
 - ☒ Test coverage
- If `bin` is negative, the function operates normally, as negative bin indices are valid.
 - ☒ Test coverage

Negative branches

- If `msg.sender` is not `FACTORY`, the function reverts with `OnlyFactory`.
 - ☒ Test coverage
- If a reentrancy attempt is detected, the `nonReentrant` modifier causes a revert.
 - ☐ Test coverage
- If `bin` is below `LOWEST_BIN` or above `HIGHEST_BIN`, the function reverts with `InvalidBinIndex(bin)`.
 - ☒ Test coverage

Function: `setPriceProvider(address newPriceProvider)` external onlyFactory

`setPriceProvider` is an `onlyFactory` function that updates the pool's mutable price provider address (the `priceProvider` state variable). However, if `IMMUTABLE_PRICE_PROVIDER` is not `address(0)`, the pool always uses `IMMUTABLE_PRICE_PROVIDER` as the actual price source, so changing `priceProvider` via this function has no effect on swap behavior. This is because `_resolvedPriceProvider()` always returns the immutable address when `IMMUTABLE_PRICE_PROVIDER != address(0)`. The function has no `nonReentrant` modifier, but since it only modifies a simple state variable without making any external calls, there is no reentrancy risk. No validation of `newPriceProvider` (e.g., whether it implements the `IPriceProvider` interface or returns the correct tokens) is performed within this function.

Inputs

- `newPriceProvider`:
 - **Control:** Factory-supplied
 - **Constraints:** No validation in the pool itself; `address(0)` is permitted (in this case, subsequent swaps may revert with `PriceProviderFailed` when `_getBidAndAskPriceX64` calls `address(0).getBidAndAskPrice()`).
 - **Impact:** Only changes the actual swap price source when `IMMUTABLE_PRICE_PROVIDER == address(0)`; otherwise, only the state variable is updated with no effect on actual behavior.

Branches and code coverage

Intended branches

- If `IMMUTABLE_PRICE_PROVIDER == address(0)`, the `priceProvider` state variable is updated and subsequent swaps use the new oracle.
 - ☒ Test coverage

- If `IMMUTABLE_PRICE_PROVIDER != address(0)`, only the `priceProvider` state variable is changed but swap behavior is unaffected, making this a dead write.

☐ Test coverage

- If `newPriceProvider == address(0)`, `address(0)` is stored in the state variable and subsequent swaps may revert with `PriceProviderFailed`.

☐ Test coverage

Negative branches

- If `msg.sender` is not `FACTORY`, the function reverts with `OnlyFactory`.

☐ Test coverage

Function: `createPool(PoolParameters calldata params)` external override returns (address pool)

`createPool` is the core factory function for deploying a new Metric OMM pool, and serves as a public interface callable by anyone. The caller provides all parameters required for pool creation via the `PoolParameters` struct, including the token pair, price provider, fee configuration, and initial bin layout. Internally, after parameter validation, scale factor computation, and bin state unpacking, the pool is deployed via `MetricOmmPoolDeployer`, and pool metadata (`poolAdmin`, `poolFeeConfig`, `poolAdminFeeDestination`, etc.) is recorded in the factory. Protocol fees are automatically applied from the factory's current global values (`spreadProtocolFeeE6`, `protocolNotionalFeeE8`), while admin fees are specified by the caller.

Inputs

- `params.token0`
 - **Control:** Arbitrary
 - **Constraints:** Cannot be `address(0)`, must differ from `token1`, must implement `IERC20Metadata.decimals()`
 - **Impact:** Set as the immutable `token0` address of the pool; used in scale factor computation
- `params.token1`
 - **Control:** Arbitrary
 - **Constraints:** Cannot be `address(0)`, must differ from `token0`, must implement `IERC20Metadata.decimals()`
 - **Impact:** Set as the immutable `token1` address of the pool; used in scale factor computation
- `params.priceProvider`

- **Control:** Arbitrary
 - **Constraints:** `IPriceProvider.getTokens()` must return `(token0, token1)` in that order; cannot be `address(0)`
 - **Impact:** Set as the pool's price oracle; fixed as immutable if `priceProviderTimelock` is `type(uint256).max`
- `params.hooks`
 - **Control:** Arbitrary
 - **Constraints:** Validated by `MetricHooks.validateHooksConfig` together with `hooksPermissions`
 - **Impact:** Hook address attached to pool operations; reverts if validation fails
- `params.hooksPermissions`
 - **Control:** Arbitrary
 - **Constraints:** Validated in `MetricHooks.validateHooksConfig` together with the `hooks` address
 - **Impact:** Determines which events the hook receives
- `params.priceProviderTimelock`
 - **Control:** Arbitrary
 - **Constraints:** No on-chain lower bound; if `type(uint256).max`, the price provider is set as immutable
 - **Impact:** Stored in the pool's `priceProviderTimelock` mapping; used as the waiting period when replacing the price provider later
- `params.admin`
 - **Control:** Arbitrary
 - **Constraints:** Cannot be `address(0)`
 - **Impact:** Set in `poolAdmin[pool]`; holds pool admin privileges (fee adjustment, pause, etc.)
- `params.initialAmount0PerShareE18`
 - **Control:** Arbitrary
 - **Constraints:** Cannot be 0
 - **Impact:** Multiplied by the scale factor and passed to the pool constructor as `initialScaledAmount0PerShareE18`; determines the initial liquidity ratio
- `params.initialAmount1PerShareE18`
 - **Control:** Arbitrary
 - **Constraints:** Cannot be 0
 - **Impact:** Multiplied by the scale factor and passed to the pool constructor as `initialScaledAmount1PerShareE18`
- `params.minimalMintableLiquidity`
 - **Control:** Arbitrary
 - **Constraints:** Cannot be 0

- **Impact:** Set as the minimum mintable liquidity threshold in the pool; used to prevent dust attacks
- `params.adminSpreadFeeE6`
 - **Control:** Arbitrary
 - **Constraints:** Must be at or below `maxAdminSpreadFeeE6`; sum with protocol fee must be at or below `maxProtocolSpreadFeeE6 + maxAdminSpreadFeeE6`
 - **Impact:** Combined with the factory's `spreadProtocolFeeE6` to determine the pool's total spread fee
- `params.adminNotionalFeeE8`
 - **Control:** Arbitrary
 - **Constraints:** Must be at or below `maxAdminNotionalFeeE8`
 - **Impact:** Combined with the factory's `protocolNotionalFeeE8` to determine the pool's total notional fee
- `params.adminFeeDestination`
 - **Control:** Arbitrary
 - **Constraints:** Cannot be `address(0)`
 - **Impact:** Stored in `poolAdminFeeDestination[pool]`; the address that receives admin fees
- `params.curBinDistFromProvidedPriceE6`
 - **Control:** Arbitrary
 - **Constraints:** Must be within the range `(-1e6, 1e6)`
 - **Impact:** Distance of the current active bin relative to the provided price; used as the reference for cumulative distance calculation during bin state unpacking
- `params.nonNegativeBinDataArray`
 - **Control:** Arbitrary
 - **Constraints:** Must not be empty; at most 5 bins per element; cumulative distance must be less than `1e6`; total bin count must be 128 or fewer
 - **Impact:** Determines the positive-side bin layout (length, buy/sell additional fees)
- `params.negativeBinDataArray`
 - **Control:** Arbitrary
 - **Constraints:** Cumulative distance must be greater than `-1e6`; total bin count must be 128 or fewer
 - **Impact:** Determines the negative-side bin layout
- `params.salt`
 - **Control:** Arbitrary
 - **Constraints:** Reusing the same salt with the same deployment parameters may cause a CREATE2 collision

- **Impact:** Used to determine the CREATE2 address; enables deterministic address computation for the deployment

Branches and code coverage

Intended branches

- When called with valid params and poolDeployer is set, the pool is deployed, metadata is recorded, and a PoolCreated event is emitted.
☒ Test coverage
- When priceProviderTimelock == type(uint256).max, the pool's price provider is set as immutable.
☒ Test coverage
- When a token has fewer than 18 decimals, the scale factor is computed correctly and passed to the pool.
☐ Test coverage
- When a token has more than 18 decimals, internalDecimals is adjusted accordingly.
☐ Test coverage
- When negativeBinDataArray is empty, the function processes normally with negBinCount == 0.
☐ Test coverage

Negative branches

- If poolDeployer == address(0), the function reverts with PoolDeployerNotSet.
☐ Test coverage
- If token0 == address(0), token1 == address(0), or token0 == token1, the function reverts with InvalidTokenConfig.
☐ Test coverage
- If admin == address(0), the function reverts with InvalidAdmin.
☐ Test coverage
- If adminFeeDestination == address(0), the function reverts with InvalidAdminFeeDestination.
☐ Test coverage
- If the price provider does not return the correct token pair, the function reverts with PriceProviderTokenMismatch.
☐ Test coverage
- If priceProvider == address(0), the function reverts with InvalidPriceProvider.

☐ Test coverage

- If `adminSpreadFeeE6` exceeds `maxAdminSpreadFeeE6`, the function reverts with `AdminFeeTooHigh`.

☐ Test coverage

- If `adminNotionalFeeE8` exceeds `maxAdminNotionalFeeE8`, the function reverts with `AdminFeeTooHigh`.

☒ Test coverage

- If the combined fee exceeds the protocol fee cap, the function reverts with `ProtocolFeeTooHigh`.

☐ Test coverage

- If `initialAmount0PerShareE18` or `initialAmount1PerShareE18` is zero, the function reverts with `InvalidInitialAmount`.

☐ Test coverage

- If `minimalMintableLiquidity` is zero, the function reverts with `InvalidMinimalMintableLiquidity`.

☐ Test coverage

- If `nonNegativeBinDataArray` is empty, the function reverts with `BinArraysEmpty`.

☒ Test coverage

- If `curBinDistFromProvidedPriceE6` $\geq 1e6$ or $\leq -1e6$, the function reverts with `BinDistanceOutOfRange`.

☒ Test coverage

- If the cumulative bin distance exceeds the allowed range, the function reverts with `BinDistanceOutOfRange`.

☒ Test coverage

- If `posBinCount` or `negBinCount` exceeds 128, the function reverts with `BinIndexRangeExceedsInt8`.

☐ Test coverage

- If a deployment is attempted to an address already occupied by the same salt, the EVM reverts due to a CREATE2 collision.

☐ Test coverage

- If hook validation fails, `MetricHooks.validateHooksConfig` causes a revert.

☒ Test coverage

Function call analysis

- `MetricOmmPoolDeployer(poolDeployer).deploy(DeployParams{...})`

- **What is controllable?** poolDeployer can only be set by the owner and is therefore trusted; all fields in DeployParams are values that have been validated earlier
- **If the return value is controllable, how is it used and how can it go wrong?** The returned pool address is used as a key in poolAdmin, poolFeeConfig, etc.; reverts on deployment failure; CREATE2 collision occurs on reuse of the same salt
- **What happens if it reverts, reenters, or does other unusual control flow?** On revert, the entire createPool call is cancelled; reentrancy is possible if an external call occurs within the new pool's constructor, but there is no nonReentrant modifier (createPool does not apply one) — during pool constructor execution, the factory state has not yet been recorded, so if reentrancy attempts to redeploy with the same salt, the second deployment will fail due to a CREATE2 collision

Function: setPoolDeployer(address _poolDeployer) external override onlyOwner

setPoolDeployer is a function that sets the MetricOmmPoolDeployer contract address used by the factory when deploying pools, and can only be set once. Restricted by onlyOwner, only the factory owner can call it; once set, it cannot be changed and any attempt results in a PoolDeployerAlreadySet error. A PoolDeployerSet event is emitted upon successful configuration, and while this value is address(0), the createPool function itself does not work.

Inputs

- _poolDeployer
 - **Control:** Owner-controlled
 - **Constraints:** Reverts if poolDeployer is already non-address(0); the validity of the input address (whether it is actually a MetricOmmPoolDeployer) is not verified on-chain
 - **Impact:** Permanently stored in the poolDeployer state variable; the deploy() function at this address will be called on subsequent createPool calls

Branches and code coverage

Intended branches

- When called with a valid address and poolDeployer == address(0), the deployer address is set and a PoolDeployerSet event is emitted.

☒ Test coverage

Negative branches

- If `poolDeployer` is already set to a non-zero address, the function reverts with `PoolDeployerAlreadySet`.
 - ☐ Test coverage
- If the caller is not the owner, the function reverts with `OwnableUnauthorizedAccount`.
 - ☐ Test coverage
- If `_poolDeployer == address(0)`, there is no on-chain validation; if stored, subsequent `createPool` calls will revert during the external call to `address(0)` rather than with `PoolDeployerNotSet`.
 - ☐ Test coverage

Function call analysis

Function: `collectTokens(address token, address to, uint256 amount)` external override `onlyOwner`

`collectTokens` is an admin function that transfers ERC-20 tokens accumulated in the factory contract to an address specified by the owner. Only the factory owner can call it due to the `onlyOwner` restriction. If `amount` is specified as 0, it provides a convenience feature to transfer the factory's entire balance. A `TokensCollected` event is emitted on successful transfer.

Inputs

- `token`
 - **Control:** Owner-controlled
 - **Constraints:** Must implement the `IERC20` interface, and `balanceOf` and `safeTransfer` calls must function correctly
 - **Impact:** Determines which token's balance to collect
- `to`
 - **Control:** Owner-controlled
 - **Constraints:** No on-chain validation (including `address(0)`); `SafeERC20.safeTransfer` may allow transfers to `address(0)` for some tokens
 - **Impact:** Transfer destination for ERC-20 tokens
- `amount`
 - **Control:** Owner-controlled

- **Constraints:** If 0, transfers the entire balance; if greater than the balance, reverts inside `safeTransfer`
- **Impact:** Determines the amount of tokens to transfer; 0 means the entire balance

Branches and code coverage

Intended branches

- When `amount > 0` and the balance is sufficient, the specified amount is transferred and an event is emitted.
☐ Test coverage
- When `amount == 0` and the balance is positive, the entire balance is transferred and an event is emitted.
☐ Test coverage
- When both `amount == 0` and the balance is zero, nothing happens and no event is emitted.
☐ Test coverage
- When `amount > 0` is specified but the balance is zero, a transfer of `amount` is attempted and reverts from the token contract.
☐ Test coverage

Negative branches

- If the caller is not the owner, the function reverts with `OwnableUnauthorizedAccount`.
☐ Test coverage
- If `amount` exceeds the token balance, `SafeERC20.safeTransfer` reverts due to insufficient balance.
☐ Test coverage
- If token is a non-standard ERC-20 with no return value, `SafeERC20` handles it and reverts if the return value is false.
☐ Test coverage

Function call analysis

- `IERC20(token).balanceOf(address(this))`
 - **What is controllable?** `token` is under owner control; a malicious token can return any value
 - **If the return value is controllable, how is it used and how can it go wrong?** The return value `balance` is used in the `amountToCollect` calculation; if a

malicious token manipulates `balanceOf` to return a value larger than the actual balance, a revert occurs during `safeTransfer` (practical impact is limited)

- **What happens if it reverts, reenters, or does other unusual control flow?**
On revert, the entire `collectTokens` call is cancelled; reentrancy is possible since there is no `nonReentrant` modifier — however, the only state change in this function is the token transfer, so double-withdrawal beyond the factory's balance would be needed to drain tokens via reentrancy, making the ERC-20 `balanceOf` reentry pattern itself low-risk

- `IERC20(token).safeTransfer(to, amountToCollect)`

- **What is controllable?** Both `to` and `amountToCollect` are under owner control
- **If the return value is controllable, how is it used and how can it go wrong?**
`SafeERC20` validates the return value, so it reverts on failure; non-standard tokens are handled correctly by `safeTransfer`
- **What happens if it reverts, reenters, or does other unusual control flow?** If a malicious token contract reenters `collectTokens` during the transfer call, the same function can be called again since there is no balance check; however, `onlyOwner` means the owner must intentionally set `to` to a malicious contract, so the practical threat is low

Function: `collectEth(address payable to, uint256 amount)` external override `onlyOwner`

`collectEth` is an admin function that transfers native ETH accumulated in the factory contract to an address specified by the owner. Only the factory owner can call it due to the `onlyOwner` restriction. If `amount` is specified as 0, the factory's entire ETH balance is transferred; on transfer failure, the call reverts with the message "ETH transfer failed". A `TokensCollected(address(0), to, amountToCollect)` event is emitted on successful transfer.

Inputs

- `to`
 - **Control:** Owner-controlled
 - **Constraints:** No on-chain validation; `address(0)` can be specified; if a contract address that cannot receive ETH is specified, the transfer will fail
 - **Impact:** ETH transfer destination; if an arbitrary contract is specified, its `receive/fallback` will be executed
- `amount`
 - **Control:** Owner-controlled

- **Constraints:** If 0, transfers the entire balance; if greater than the balance, the low-level call fails and reverts
- **Impact:** Determines the amount of ETH to transfer; 0 means the entire balance

Branches and code coverage

Intended branches

- When `amount > 0` and the ETH balance is sufficient, the specified amount is transferred and an event is emitted.
☐ Test coverage
- When `amount == 0` and the ETH balance is positive, the entire balance is transferred and an event is emitted.
☐ Test coverage
- When both `amount == 0` and the ETH balance is zero, nothing happens and no event is emitted.
☐ Test coverage

Negative branches

- If the caller is not the owner, the function reverts with `OwnableUnauthorizedAccount`.
☐ Test coverage
- If `to` is a contract that cannot receive ETH, the low-level call fails and the function reverts with `"ETH transfer failed"`.
☐ Test coverage
- If `amount` exceeds the factory's ETH balance, the call fails and the function reverts with `"ETH transfer failed"`.
☐ Test coverage
- If `to` is a contract whose `receive` function reverts, the call fails and the function reverts with `"ETH transfer failed"`.
☐ Test coverage

Function call analysis

- `to.call{value: amountToCollect}("")`
 - **What is controllable?** Both `to` and `amountToCollect` are under owner control; if `to` is an arbitrary contract address, that contract's code will be executed

- **If the return value is controllable, how is it used and how can it go wrong?** If the success return value is false, reverts with `require(success, "ETH transfer failed")`; since a low-level call is used without the 2300 gas stipend, the receiving contract can execute complex logic
- **What happens if it reverts, reenters, or does other unusual control flow?**
Reentrancy risk: `collectEth` has no `nonReentrant` modifier; if to is a malicious contract, it can recursively call `collectEth` from the ETH receive callback; however, `onlyOwner` requires the owner to intentionally set to to a malicious contract, and since `amountToCollect` is determined based on the balance, there is a chance the balance has already decreased by the second reentrant call — however, since balance is read directly from `address(this).balance` rather than from a state variable, if the factory's balance has not yet been updated at the point of reentry, the same amount could be transferred twice

Function: `setFeeCaps(uint24 newMaxProtocolSpreadFeeE6, uint24 newMaxAdminSpreadFeeE6, uint24 newMaxProtocolNotionalFeeE8, uint24 newMaxAdminNotionalFeeE8)` external override `onlyOwner`

`setFeeCaps` is a function that allows the protocol owner to adjust the maximum fee caps (`maxProtocol*`, `maxAdmin*`) applied to future pool creation and fee changes. It is restricted to the factory owner via the `onlyOwner` modifier, and reverts if any new cap exceeds the hardcoded absolute upper bounds (`HARD_MAX_SPREAD_FEE_E6 = 200_000`, `HARD_MAX_NOTIONAL_FEE_E8 = 1_000_000`). This function does not retroactively change fees for existing pools; it affects the upper bounds for subsequent fee-related operations such as `createPool`, `setPoolProtocolFee`, and `setPoolAdminFees`. A `FeeCapsUpdated` event is emitted upon completion.

Inputs

- `newMaxProtocolSpreadFeeE6`
 - **Control:** Owner-controlled
 - **Constraints:** Must be at or below `HARD_MAX_SPREAD_FEE_E6 (200_000)`; exceeding it reverts with `FeeCapsExceedHardLimit`
 - **Impact:** Updates `maxProtocolSpreadFeeE6`; used as the protocol spread fee cap in subsequent calls to `setPoolProtocolFee`
- `newMaxAdminSpreadFeeE6`
 - **Control:** Owner-controlled
 - **Constraints:** Must be at or below `HARD_MAX_SPREAD_FEE_E6 (200_000)`; exceeding it reverts with `FeeCapsExceedHardLimit`
 - **Impact:** Updates `maxAdminSpreadFeeE6`; used as the admin spread fee cap in subsequent calls to `createPool` and `setPoolAdminFees`

- `newMaxProtocolNotionalFeeE8`
 - **Control:** Owner-controlled
 - **Constraints:** Must be at or below `HARD_MAX_NOTIONAL_FEE_E8` (`1_000_000`); exceeding it reverts with `FeeCapsExceedHardLimit`
 - **Impact:** Updates `maxProtocolNotionalFeeE8`; used as the protocol notional fee cap in subsequent calls to `setPoolProtocolFee`
- `newMaxAdminNotionalFeeE8`
 - **Control:** Owner-controlled
 - **Constraints:** Must be at or below `HARD_MAX_NOTIONAL_FEE_E8` (`1_000_000`); exceeding it reverts with `FeeCapsExceedHardLimit`
 - **Impact:** Updates `maxAdminNotionalFeeE8`; used as the admin notional fee cap in subsequent calls to `createPool` and `setPoolAdminFees`

Branches and code coverage

Intended branches

- When all new caps are at or below the hard limits, all four cap state variables are updated and a `FeeCapsUpdated` event is emitted.
 - ☒ Test coverage
- When called to lower a cap, fees on existing pools are not retroactively changed; only subsequent operations are constrained by the new cap.
 - ☒ Test coverage
- When a cap is set exactly equal to the hard limit, the function processes normally.
 - ☒ Test coverage

Negative branches

- If the caller is not the owner, the function reverts with `OwnableUnauthorizedAccount`.
 - ☒ Test coverage
- If `newMaxProtocolSpreadFeeE6` exceeds `HARD_MAX_SPREAD_FEE_E6`, the function reverts with `FeeCapsExceedHardLimit`.
 - ☒ Test coverage
- If `newMaxAdminSpreadFeeE6` exceeds `HARD_MAX_SPREAD_FEE_E6`, the function reverts with `FeeCapsExceedHardLimit`.
 - ☐ Test coverage
- If `newMaxProtocolNotionalFeeE8` exceeds `HARD_MAX_NOTIONAL_FEE_E8`, the function reverts with `FeeCapsExceedHardLimit`.
 - ☒ Test coverage

- If `newMaxAdminNotionalFeeE8` exceeds `HARD_MAX_NOTIONAL_FEE_E8`, the function reverts with `FeeCapsExceedHardLimit`.
 - ☐ Test coverage
- When a cap is set to 0, there is no on-chain rejection; however, subsequent calls to `createPool` or `setPoolProtocolFee` may fail with `ProtocolFeeTooHigh` or `AdminFeeTooHigh`, effectively making fee configuration impossible.
 - ☐ Test coverage

Function call analysis

Function: `setPoolProtocolFee(address pool, uint24 newProtocolSpreadFeeE6, uint24 newProtocolNotionalFeeE8)` external override `onlyOwner` `nonReentrant`

`setPoolProtocolFee` is a function that allows the owner to update the protocol fees (spread and notional) applied to a specific pool. The `onlyOwner` and `nonReentrant` modifiers are applied. Before updating the fees, accrued fees under the existing fee configuration are first settled (`collectFees`). If the protocol spread/notional fees exceed the current `maxProtocol*` cap, the function reverts. When fees are changed, if the admin fees are higher than the current `maxAdmin*` cap they are clipped down to the cap level, and finally the pool's internal fees are updated via `IMetricOmmPoolFactoryActions.setPoolFees`.

Inputs

- `pool`
 - **Control:** Owner-controlled
 - **Constraints:** Whether the address is actually a valid pool is not verified on-chain; if an arbitrary address is passed, the `IMetricOmmPoolCollectFees.collectFees` and `IMetricOmmPoolFactoryActions.setPoolFees` calls may fail
 - **Impact:** Determines which pool's fee configuration is changed; also affects factory state such as `poolFeeConfig[pool]` and `poolAdminFeeDestination[pool]`
- `newProtocolSpreadFeeE6`
 - **Control:** Owner-controlled
 - **Constraints:** Must be at or below `maxProtocolSpreadFeeE6`; exceeding it reverts with `ProtocolFeeTooHigh`
 - **Impact:** Updates `poolFeeConfig.protocolSpreadFeeE6` for the pool, which is summed with the admin spread fee and applied to the pool

- `newProtocolNotionalFeeE8`
 - **Control:** Owner-controlled
 - **Constraints:** Must be at or below `maxProtocolNotionalFeeE8`; exceeding it reverts with `ProtocolFeeTooHigh`
 - **Impact:** Updates `poolFeeConfig.protocolNotionalFeeE8` for the pool, which is summed with the admin notional fee and applied to the pool

Branches and code coverage

Intended branches

- When the new fees are within the protocol cap and admin fees are within the `maxAdmin` cap, accrued fees are settled, the new fees are stored, `setPoolFees` is called, and an event is emitted.
 - ☒ Test coverage
- If the current admin spread fee (`c.adminSpreadFeeE6`) exceeds `maxAdminSpreadFeeE6`, it is clipped down to `maxAdminSpreadFeeE6`.
 - ☒ Test coverage
- If the current admin notional fee (`c.adminNotionalFeeE8`) exceeds `maxAdminNotionalFeeE8`, it is clipped down to `maxAdminNotionalFeeE8`.
 - ☒ Test coverage
- When the new protocol fee is set to 0, the pool is configured to operate with no protocol fee.
 - ☐ Test coverage

Negative branches

- If the caller is not the owner, the function reverts with `OwnableUnauthorizedAccount`.
 - ☒ Test coverage
- If `newProtocolSpreadFeeE6` exceeds `maxProtocolSpreadFeeE6`, the function reverts with `ProtocolFeeTooHigh`.
 - ☒ Test coverage
- If `newProtocolNotionalFeeE8` exceeds `maxProtocolNotionalFeeE8`, the function reverts with `ProtocolFeeTooHigh`.
 - ☒ Test coverage
- If `pool` is not a valid pool address, the `collectFees` or `setPoolFees` call fails and reverts.
 - ☐ Test coverage
- If `pool` is a malicious contract attempting reentrancy, the `nonReentrant` modifier blocks it.

- ☐ Test coverage
- If `poolAdminFeeDestination[pool]` is `address(0)` because the pool admin has not set it, the `collectFees` call may fail.
- ☐ Test coverage

Function call analysis

- `IMetricOmmPoolCollectFees(pool).collectFees(c.protocolSpreadFeeE6, c.adminSpreadFeeE6, c.protocolNotionalFeeE8, c.adminNotionalFeeE8, poolAdminFeeDestination[pool])`
 - **What is controllable?** The `pool` address is owner-controlled; the `c.*` values are the existing configuration stored in the factory; `poolAdminFeeDestination[pool]` is the value set by the admin
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value; this call distributes fees accrued in the pool to the protocol and admin; if the call fails, the entire fee change is reverted — if the pool blocks `collectFees`, fee changes may become permanently impossible
 - **What happens if it reverts, reenters, or does other unusual control flow?** A revert cancels the entire `setPoolProtocolFee` call; the `nonReentrant` modifier prevents the pool from re-entering the factory; however, if `pool` is an arbitrary address, a malicious contract implementing the `collectFees` selector could attempt reentrancy, but `nonReentrant` blocks it
 - `IMetricOmmPoolFactoryActions(pool).setPoolFees(c.protocolSpreadFeeE6 + c.adminSpreadFeeE6, c.protocolNotionalFeeE8 + c.adminNotionalFeeE8)`
 - **What is controllable?** The summed fee values are determined by factory state and the new inputs; the `pool` address is owner-controlled
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value; the summed fees are recorded as the pool's internal spread/notional fee state; the possibility of overflow in the `uint24` addition should be checked (Solidity 0.8.x applies default overflow checks, but it should be confirmed that values are guaranteed to be below `type(uint24).max`)
 - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, the fee settlement already performed by `collectFees` cannot be undone, but the factory's `poolFeeConfig` state change is cancelled — a consistency issue may arise where fee settlement completes but the new fee configuration fails to apply; `nonReentrant` blocks reentrancy

Function: setDefaultSpreadProtocolFeeE6(uint24 newFeeE6)

Updates the **default protocol spread fee applied when creating new pools** in the factory. This value is passed to the pool contract as `protocolSpreadFeeE6` when `createPool` is called, so it affects all pools deployed after the change. Conversely, the fee rate of already-deployed pools is not changed; to update existing pools, `setPoolProtocolFee` must be called separately. The `onlyOwner` modifier is applied, so only the factory owner can call this function.

Inputs

- `newFeeE6`
 - **Control:** Owner-controlled
 - **Constraints:** Must satisfy the condition `newFeeE6 <= maxProtocolSpreadFeeE6`. Since `maxProtocolSpreadFeeE6` cannot exceed the hard cap `HARD_MAX_SPREAD_FEE_E6` (200,000 = 20%), the effective maximum allowed value is 20%.
 - **Impact:** `spreadProtocolFeeE6` in factory storage is updated, and is reflected in the `protocolSpreadFeeE6` of new pools when `createPool` is subsequently called. The spread fee rate of already-deployed pools is not affected.

Branches and code coverage

Intended branches

- When `newFeeE6 <= maxProtocolSpreadFeeE6`, `spreadProtocolFeeE6` is updated to `newFeeE6` and a `SpreadProtocolFeeDefaultUpdated(oldFeeE6, newFeeE6)` event is emitted.
 - ☒ Test coverage
- When `newFeeE6 == 0` (fee removal), the function executes normally and resets the default to 0.
 - ☐ Test coverage
- When `newFeeE6 == maxProtocolSpreadFeeE6` (boundary value), the function executes normally.
 - ☐ Test coverage

Negative branches

- If `newFeeE6 > maxProtocolSpreadFeeE6`, the function reverts with `ProtocolFeeTooHigh`.
 - ☒ Test coverage
- If `msg.sender != owner()`, the function reverts with `OwnableUnauthorizedAccount`.
 - ☐ Test coverage

Function: setDefaultProtocolNotionalFeeE8(uint24 newFeeE8)

Updates the **default protocol notional fee applied when creating new pools** in the factory. This value is passed to the new pool's protocolNotionalFeeE8 when createPool is called, and affects all subsequently created pools. As with the spread fee, the fee rate of already-deployed pools is not changed; to update existing pools, setPoolProtocolFee must be called separately. The onlyOwner modifier is applied, so only the factory owner can call this function.

Inputs

- newFeeE8
 - **Control:** Owner-controlled
 - **Constraints:** Must satisfy the condition newFeeE8 <= maxProtocolNotionalFeeE8. Since maxProtocolNotionalFeeE8 cannot exceed the hard cap HARD_MAX_NOTIONAL_FEE_E8 (1,000,000 = 1%), the effective maximum allowed value is 1%. (1e8 = 100%, 1,000,000 = 1%)
 - **Impact:** protocolNotionalFeeE8 in factory storage is updated, and is reflected in the notional fee rate of new pools when createPool is subsequently called.

Branches and code coverage

Intended branches

- When newFeeE8 <= maxProtocolNotionalFeeE8, protocolNotionalFeeE8 is updated to newFeeE8 and a ProtocolNotionalFeeDefaultUpdated(oldFeeE8, newFeeE8) event is emitted.
 - ☒ Test coverage
- When newFeeE8 == 0 (notional fee removal), the function executes normally.
 - ☐ Test coverage
- When newFeeE8 == maxProtocolNotionalFeeE8 (boundary value), the function executes normally.
 - ☐ Test coverage

Negative branches

- If newFeeE8 > maxProtocolNotionalFeeE8, the function reverts with ProtocolFeeTooHigh.
 - ☒ Test coverage
- If msg.sender != owner(), the function reverts with OwnableUnauthorizedAccount.

☐ Test coverage

Function: collectPoolFees(address pool)

Forcibly settles (collects) accrued protocol and admin fees for a specific pool. Called by the factory owner to withdraw fees at an arbitrary point in time; both `onlyOwner` and `nonReentrant` modifiers are applied. Internally, it makes an external call to `IMetricOmmPoolCollectFees(pool).collectFees` using the current fee rate configuration stored in the factory (`poolFeeConfig[pool]`). It distributes the pool's accrued balance to the protocol (factory address) and the admin (`poolAdminFeeDestination[pool]`), without modifying the factory's fee rate configuration itself.

Inputs

- `pool`
 - **Control:** Owner-controlled
 - **Constraints:** No on-chain constraints. Even if an invalid address or malicious contract is passed, `poolFeeConfig[pool]` is read before any revert occurs.
 - **Impact:** Fees are settled from the corresponding pool contract. If `pool` is an address not registered with the factory, `poolFeeConfig[pool]` will be the default value (all zeros) and `poolAdminFeeDestination[pool]` will be `address(0)`, which may cause a revert inside the pool when `collectFees` is called.

Branches and code coverage

Intended branches

- When `pool` is a valid factory-registered pool, fees are settled at the fee rate stored in `poolFeeConfig[pool]` and tokens are transferred to the protocol and admin destinations.
 - ☐ Test coverage
- When there are no accrued fees, `collectFees` is called but no actual token movement occurs; only an event is emitted.
 - ☐ Test coverage

Negative branches

- If `msg.sender` is not the owner, the function reverts with `OwnableUnauthorizedAccount`.
 - ☐ Test coverage

- If `pool` is `address(0)` or an EOA, the function reverts during the call to `IMetricOmmPoolCollectFees(pool).collectFees`.
 - ☐ Test coverage
- If `pool` is a contract that implements `OnlyFactory` but is not registered with this factory, the pool contract may revert with a `msg.sender != FACTORY` check.
 - ☐ Test coverage
- If a reentrancy attempt is detected, the `nonReentrant` modifier blocks it.
 - ☐ Test coverage

Function call analysis

- `IMetricOmmPoolCollectFees(pool).collectFees(c.protocolSpreadFeeE6, c.adminSpreadFeeE6, c.protocolNotionalFeeE8, c.adminNotionalFeeE8, poolAdminFeeDestination[pool])`
 - **What is controllable?** The `pool` address (owner-controlled); the fee rate parameters passed are read from factory storage `poolFeeConfig[pool]` so they cannot be directly controlled, but the owner can change those values via `setPoolProtocolFee`. `poolAdminFeeDestination[pool]` cannot be indirectly controlled by the owner, but the pool admin can change it via `setPoolAdminFeeDestination`.
 - **If the return value is controllable, how is it used and how can it go wrong?** `collectFees` has no return value. The actual fee distribution logic is handled internally by the pool, calculated based on the fee rates stored in the factory and the accumulated balance inside the pool. If there is a mismatch between fee rates in the factory and the pool (e.g., factory settings changed without calling `setPoolProtocolFee`), the distribution ratio may differ from the intended configuration.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A revert inside the pool rolls back the entire transaction. If the pool contains malicious code, it may attempt reentrancy, but this is blocked by `nonReentrant` (based on EIP-1153 transient storage). Since `pool` can be an arbitrary external address, arbitrary code can be executed via this external call; care should be taken in scenarios where the owner passes an untrusted `pool` address.

Function: `protocolPausePool(address pool)`

Pauses a specific pool at the protocol level (pause level 2). Only the factory owner can call this function (`onlyOwner`, `nonReentrant`). Before execution, it reads the pool's current pause level via `EXTSLOAD` through `PoolStateLibrary._slot0(pool)` and validates that the transition is valid.

Allowed transitions are 0 (active) → 2 or 1 (admin paused) → 2; any other state reverts. When at pause level 2, ordinary user transactions and liquidity changes are blocked, and the pool can only be unpaused by the owner via `protocolUnpausePool`.

Inputs

- `pool`
 - **Control:** Owner-controlled
 - **Constraints:** Registration with the factory is not separately verified on-chain. However, since `PoolStateLibrary._slot0(pool)` uses an `EXTSLOAD` call, the function will revert if `pool` does not implement `IExtslload`.
 - **Impact:** The pool's pause level transitions to 2. On a paused pool, user interactions such as swaps and adding/removing liquidity are blocked.

Branches and code coverage

Intended branches

- When the current pause level is 0 (active state), the pause level transitions from 0 to 2 and a `PauseLevelUpdated(0, 2)` event is emitted from the pool.
 - ☒ Test coverage
- When the current pause level is 1 (admin-paused state), the pause level is upgraded from 1 to 2 and a `PauseLevelUpdated(1, 2)` event is emitted.
 - ☒ Test coverage

Negative branches

- If the current pause level is already 2, the function reverts with `InvalidPauseTransition(2, 2)`.
 - ☒ Test coverage
- If the current pause level is 3 or higher (undefined), the function reverts with `InvalidPauseTransition(cur, 2)`.
 - ☐ Test coverage
- If `msg.sender` is not the owner, the function reverts with `OwnableUnauthorizedAccount`.
 - ☐ Test coverage
- If a reentrancy attempt is detected, the `nonReentrant` modifier blocks it.
 - ☐ Test coverage
- If `pool` does not implement `EXTSLOAD`, the function reverts during the `PoolStateLibrary._slot0` call.
 - ☐ Test coverage

Function call analysis

- `PoolStateLibrary._slot0(pool)`
 - **What is controllable?** The pool address (owner-controlled). Reads slot 0 of the pool via `EXTSLOAD`.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned `cur` (pauseLevel, uint8) is used directly in branching conditions. Since it is read from pool storage, the caller cannot manipulate it, but if the pool contract is maliciously implemented, it could return an arbitrary value. In a normal Metric OMM pool, the pauseLevel should be one of 0, 1, or 2. Since `IExtload(pool).extsload(bytes32(uint256(0)))` is called internally, if pool implements `extsload` incorrectly, it may revert after consuming gas.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A revert inside `_slot0` rolls back the entire transaction. The reentrancy risk is low since it is a read-only call; a malicious pool cannot have side effects that modify state during the `extsload` call (view call).
- `IMetricOmmPoolFactoryActions(pool).setPause(2)`
 - **What is controllable?** The pause level value (fixed at 2). The pool address is controlled by the owner.
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value. Inside the pool, after verifying `msg.sender == FACTORY`, the pauseLevel field in slot 0 is updated to 2 and a `PauseLevelUpdated` event is emitted.
 - **What happens if it reverts, reenters, or does other unusual control flow?** The pool may revert with an `OnlyFactory` error (if pool address is incorrect) or an `InvalidPauseLevel` error. A malicious pool may attempt reentrancy, but this is blocked by `nonReentrant`. If another transaction changes the pause level between the `PoolStateLibrary._slot0` read and the `setPause` call (TOCTOU), since the `cur` check has already passed, `setPause` could trigger an unexpected state transition.

Function: `protocolUnpausePool(address pool)`

Downgrades a protocol-level paused pool (pause level 2) to the admin-paused state (pause level 1). Only the factory owner can call this function (`onlyOwner`, `nonReentrant`). Before executing, the current pause level is read via `PoolStateLibrary._slot0(pool)`, and the transition is only allowed when `cur == 2`. After the downgrade, the pool is in pause level 1 (admin-paused) state; full activation (level 0) can only be achieved through the admin's `unpausePool`. This creates a two-step unpause structure in which the owner can lift the protocol emergency halt, but the pool admin must make the final decision to resume.

Inputs

- pool
 - **Control:** Owner-controlled
 - **Constraints:** Registration is not separately verified on-chain. If pool does not implement IExtslod, the call to PoolStateLibrary._slot0 will revert.
 - **Impact:** The pause level of the target pool is downgraded from 2 → 1. At pause level 1, user transactions may still be restricted, and the admin must call unpausePool to transition to level 0.

Branches and code coverage

Intended branches

- When the current pause level is 2 (protocol-paused state), the pause level transitions from 2 to 1 and the PauseLevelUpdated(2, 1) event is emitted from the pool.

☒ Test coverage

Negative branches

- If the current pause level is 0 (active state), the function reverts with InvalidPauseTransition(0, 1).

☒ Test coverage

- If the current pause level is 1 (admin-paused state), the function reverts with InvalidPauseTransition(1, 1).

☐ Test coverage

- If the current pause level is 3 or higher (undefined), the function reverts with InvalidPauseTransition(cur, 1).

☐ Test coverage

- If msg.sender is not the owner, the function reverts with OwnableUnauthorizedAccount.

☐ Test coverage

- If a reentrancy attempt is detected, the nonReentrant modifier blocks it.

☐ Test coverage

- If pool does not implement EXTSLLOAD, the function reverts during the PoolStateLibrary._slot0 call.

☐ Test coverage

Function call analysis

- PoolStateLibrary._slot0(pool)

- **What is controllable?** The pool address (owner-controlled). Reads slot 0 of the pool via EXTSLOAD.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned cur (pauseLevel, uint8) is directly used in the cur != 2 condition check. Since it is read from pool storage, the caller cannot directly manipulate it. However, if another transaction changes the pause level between the PoolStateLibrary._slot0 read and the setPause call (across blocks), a TOCTOU condition can arise. In such a case, if protocolPausePool had re-set level 2 or the admin had changed the level via pausePool, an unintended state transition could occur.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If a revert occurs inside _slot0, the entire transaction is rolled back.
- IMetricOmmPoolFactoryActions(pool).setPause(1)
 - **What is controllable?** The pause level value (fixed at 1). The pool address is controlled by the owner.
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value. Inside the pool, after verifying msg.sender == FACTORY, the pauseLevel field in slot 0 is updated to 1 and the PauseLevelUpdated event is emitted. Notably, protocolUnpausePool only downgrades to level 1 (admin-paused) and does not directly transition to fully active (level 0). This is an intentional design decision for a two-step unpause structure for security purposes; however, if the admin is offline or refuses to cooperate, the pool may be permanently stuck at pause level 1.
 - **What happens if it reverts, reenters, or does other unusual control flow?** The pool may revert with OnlyFactory or InvalidPauseLevel. A malicious pool may attempt reentrancy, but this is blocked by nonReentrant.

Function: setPoolAdminFees(address pool, uint24 newAdminSpreadFeeE6, uint24 newAdminNotionalFeeE8)

The pool admin **changes the admin spread fee and notional fee** applied to their pool. The onlyPoolAdmin(pool) and nonReentrant modifiers are applied. A key design pattern is that fees must always be **settled (collectFees) using the current fee rate before applying the new rate**. This ensures that fees accumulated during the previous period are settled at the correct ratio. Finally, the sum of protocol + admin fees is reflected in the pool contract via IMetricOmmPoolFactoryActions(pool).setPoolFees.

Inputs

- pool
- **Control:** Pool admin-controlled

- **Constraints:** `poolAdmin[pool] == msg.sender` must hold; otherwise, reverts with `NotPoolAdmin`.
 - **Impact:** The admin fee rates for the target pool are changed, and `collectFees` and `setPoolFees` are called in sequence internally.
- `newAdminSpreadFeeE6`
 - **Control:** Pool admin-controlled
 - **Constraints:** `newAdminSpreadFeeE6 <= maxAdminSpreadFeeE6` must hold. `maxAdminSpreadFeeE6` is set by the factory owner and cannot exceed the hard cap (`HARD_MAX_SPREAD_FEE_E6 = 200,000 = 20%`).
 - **Impact:** `poolFeeConfig[pool].adminSpreadFeeE6` is updated, and the total spread fee set on the pool becomes `protocolSpreadFeeE6 + newAdminSpreadFeeE6`.
- `newAdminNotionalFeeE8`
 - **Control:** Pool admin-controlled
 - **Constraints:** `newAdminNotionalFeeE8 <= maxAdminNotionalFeeE8` must hold. `maxAdminNotionalFeeE8` is set by the factory owner and cannot exceed the hard cap (`HARD_MAX_NOTIONAL_FEE_E8 = 1,000,000 = 1%`).
 - **Impact:** `poolFeeConfig[pool].adminNotionalFeeE8` is updated, and the total notional fee set on the pool becomes `protocolNotionalFeeE8 + newAdminNotionalFeeE8`.

Branches and code coverage

Intended branches

- When all validations pass, the function executes `collectFees`, updates `poolFeeConfig`, and calls `setPoolFees`, emitting `PoolAdminSpreadFeeUpdated` and `PoolAdminNotionalFeeUpdated` events.
 - ☒ Test coverage
- When `newAdminSpreadFeeE6 == 0` or `newAdminNotionalFeeE8 == 0` (removing the admin fee), the function executes normally.
 - ☒ Test coverage
- When `newAdminSpreadFeeE6 == maxAdminSpreadFeeE6` and `newAdminNotionalFeeE8 == maxAdminNotionalFeeE8` (boundary values), the function should be allowed to proceed.
 - ☐ Test coverage
- When there are no accumulated fees, `collectFees` performs a zero-settlement and only the fee rates are updated.
 - ☐ Test coverage

Negative branches

- If `msg.sender` is not the pool admin, the function reverts with `NotPoolAdmin`.
 - ☐ Test coverage
- If `newAdminSpreadFeeE6` exceeds `maxAdminSpreadFeeE6`, the function reverts with `AdminFeeTooHigh`.
 - ☐ Test coverage
- If `newAdminNotionalFeeE8` exceeds `maxAdminNotionalFeeE8`, the function reverts with `AdminFeeTooHigh`.
 - ☐ Test coverage
- If a reentrancy attempt is detected, the `nonReentrant` modifier blocks it.
 - ☐ Test coverage
- If `pool` is an invalid address, the function reverts during the `collectFees` external call.
 - ☐ Test coverage

Function call analysis

- `IMetricOmmPoolCollectFees(pool).collectFees(c.protocolSpreadFeeE6, c.adminSpreadFeeE6, c.protocolNotionalFeeE8, c.adminNotionalFeeE8, poolAdminFeeDestination[pool])`
 - **What is controllable?** The pool address and the current fee rate configuration (`poolFeeConfig[pool]`) are used. Since `poolFeeConfig` at call time holds the old value (before the change), the purpose is to settle accumulated fees from the previous period at the old ratio. `poolAdminFeeDestination[pool]` can be changed by the pool admin via `setPoolAdminFeeDestination`.
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value. Inside `collectFees`, fee balances accumulated in the pool are distributed to the protocol and admin destinations. If the fee rates are inconsistent with the internal accumulation logic of the pool, the distribution ratio may differ from the intended one. A scenario is possible in which an attacker deliberately induces large trades just before the admin changes the fee rate, causing settlement to occur while the admin fee ratio is low (front-running).
 - **What happens if it reverts, reenters, or does other unusual control flow?** If `collectFees` reverts, the fee rate change is also rolled back. Even if a malicious pool attempts reentrancy, it is blocked by `nonReentrant`. However, since token transfers occur inside `collectFees`, if `adminFeeDestination` is a contract that rejects receipt (reverts), `collectFees` will revert, making it impossible to change the fee rate at all – a DOS condition.
- `IMetricOmmPoolFactoryActions(pool).setPoolFees(c.protocolSpreadFeeE6 + c.adminSpreadFeeE6, c.protocolNotionalFeeE8 + c.adminNotionalFeeE8)`

- **What is controllable?** The combined fee rates are the values after the poolFeeConfig update ($c.protocolSpreadFeeE6 + newAdminSpreadFeeE6$, $c.protocolNotionalFeeE8 + newAdminNotionalFeeE8$). Even if each component individually passes its cap check, the combined value can reach $maxProtocolSpreadFeeE6 + maxAdminSpreadFeeE6$.
- **If the return value is controllable, how is it used and how can it go wrong?** No return value. The spreadFeeE6 and notionalFeeE8 fields in slot 0 of the pool contract are updated and an event is emitted. Overflow risk from uint24 addition: $c.protocolSpreadFeeE6 + c.adminSpreadFeeE6$ is a sum of two uint24 values, so the maximum is 400,000 (40%), which is within the uint24 range (approximately 16.7M). Overflow risk is practically nonexistent.
- **What happens if it reverts, reenters, or does other unusual control flow?** If the pool reverts with OnlyFactory, the poolFeeConfig update is rolled back. This means a state where fees have already been settled via collectFees but the fee rate change fails cannot occur (full transaction rollback).

Function: setPoolAdminFeeDestination(address pool, address newAdminFeeDestination) external override nonReentrant onlyPoolAdmin(pool)

A function that changes the recipient address for pool admin fees. Only poolAdmin[pool] can call it, and the zero address is explicitly rejected. The state change only updates the factory's poolAdminFeeDestination[pool] mapping; the new address will be used on the next call to collectFees. The PoolAdminFeeDestinationUpdated event is emitted.

Inputs

- pool
 - **Control:** Pool admin-controlled
 - **Constraints:** The validity of the pool address is not separately verified on-chain. If an address not registered in the factory is passed, poolAdmin[pool] is the zero address, so it reverts with NotPoolAdmin.
 - **Impact:** Determines which poolAdminFeeDestination slot this parameter selects.
- newAdminFeeDestination
 - **Control:** Pool admin-controlled
 - **Constraints:** If address(0), reverts with InvalidAdminFeeDestination. Any other address is allowed.
 - **Impact:** Overwrites poolAdminFeeDestination[pool]. On subsequent calls to collectFees, admin fees will be sent to this address. If set to a malicious contract address, reentrancy or receipt failure may occur during fee collection.

Branches and code coverage

Intended branches

- When `newAdminFeeDestination != address(0)` and the caller is `poolAdmin[pool]`, `poolAdminFeeDestination[pool]` is updated and a `PoolAdminFeeDestinationUpdated` event is emitted.

☐ Test coverage

Negative branches

- If `msg.sender` is not the pool admin, the function reverts with `NotPoolAdmin`.

☐ Test coverage

- If `newAdminFeeDestination == address(0)`, the function reverts with `InvalidAdminFeeDestination`.

☐ Test coverage

- When `pool` is an address never registered in the factory, `poolAdmin[pool]` is `address(0)`, causing the function to revert with `NotPoolAdmin`.

☐ Test coverage

Function call analysis

- (Event emission `PoolAdminFeeDestinationUpdated` — not an external call)
 - Only an event is emitted with no external contract calls, so there is no reentrancy risk.

Function: `setPoolBinAdditionalFees(address pool, int8 bin, uint16 addFeeBuyE6, uint16 addFeeSellE6)` external override nonReentrant onlyPoolAdmin(pool)

A function that sets additional buy/sell spread fees for a specific bin in a specific pool. Only `poolAdmin[pool]` can call it. The factory does not independently validate the parameters and directly delegates to the pool contract's `setBinAdditionalFees`. Bin index range and fee cap validation are performed on the pool side. Note that this function does not apply the factory-level fee caps (such as `maxAdminSpreadFeeE6`).

Inputs

- `pool`

- **Control:** Pool admin-controlled
 - **Constraints:** Must be a pool registered in the factory, and the caller must be the admin of that pool.
 - **Impact:** Determines which pool's bin fees to change.
- bin
 - **Control:** Pool admin-controlled
 - **Constraints:** The pool contract (MetricOmmPool) validates whether the bin is within the allowed range (lowestBin ~ highestBin). If out of range, the pool side reverts.
 - **Impact:** Specifies the particular bin whose fees are to be changed. The buy/sell fees for that bin are updated.
- addFeeBuyE6
 - **Control:** Pool admin-controlled
 - **Constraints:** There is no cap validation at the factory level. The pool contract may validate internally.
 - **Impact:** Sets the additional buy-side spread fee for that bin. Can be set up to the uint16 maximum (65535 = 6.5535%), and is applied in addition to the base spread fee.
- addFeeSellE6
 - **Control:** Pool admin-controlled
 - **Constraints:** There is no cap validation at the factory level.
 - **Impact:** Sets the additional sell-side spread fee for that bin. The same risks as the buy side apply.

Branches and code coverage

Intended branches

- When called with a valid bin index and normal fee values, setBinAdditionalFees in the pool contract succeeds and the BinAdditionalFeesUpdated event is emitted from the pool.

☒ Test coverage

Negative branches

- If msg.sender is not the pool admin, the function reverts with NotPoolAdmin.

☒ Test coverage

- If bin is outside the pool's allowed range (lowestBin to highestBin), the pool contract reverts.

☒ Test coverage

- If pool is an invalid or unregistered address, the external call fails.

□ Test coverage

Function call analysis

- `IMetricOmmPoolFactoryActions(pool).setBinAdditionalFees(bin, addFeeBuyE6, addFeeSellE6)`
 - **What is controllable?** All three parameters — `bin`, `addFeeBuyE6`, and `addFeeSellE6` — are controlled by the pool admin. The `pool` address is also specified by the admin.
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value. The factory trusts the pool to validate internally. Since the factory does not check the caps for `addFeeBuyE6` and `addFeeSellE6`, if pool-side validation is missing or bypassed, abnormally high per-bin fees could be set.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the pool contract reverts, the factory call reverts as well. If the pool is a malicious contract, reentrancy is possible, but the `nonReentrant` modifier prevents re-entry into the factory. However, cross-contract reentrancy via calls to external contracts through the pool is not defended against.

Function: `pausePool(address pool) external override nonReentrant only-PoolAdmin(pool)`

A function by which the pool admin transitions a pool to the paused state (admin level 1). Only `poolAdmin[pool]` can call it, and the current pause state must be exactly 0 (active). The current `pauseLevel` of the pool is read via `PoolStateLibrary._slot0`, and only the valid transition (`0 → 1`) is allowed. If called in an invalid state, it reverts with `InvalidPauseTransition`. The protocol-level pause (level 2) cannot be released through this function.

Inputs

- `pool`
 - **Control:** Pool admin-controlled
 - **Constraints:** Must be a pool registered in the factory, the caller must be the admin of that pool, and the current `pauseLevel` must be exactly 0.
 - **Impact:** Determines which pool to pause.

Branches and code coverage

Intended branches

- When `pauseLevel == 0` and the caller is the pool admin, the pool's `pauseLevel` is changed to 1 and the `PauseLevelUpdated` event is emitted from the pool.

☒ Test coverage

Negative branches

- If `msg.sender` is not the pool admin, the function reverts with `NotPoolAdmin`.

☐ Test coverage

- If the pool is already in admin-paused state (`pauseLevel == 1`), the function reverts with `InvalidPauseTransition(1, 1)`.

☒ Test coverage

- If the pool is in protocol-paused state (`pauseLevel == 2`), the function reverts with `InvalidPauseTransition(2, 1)`, as the admin cannot transition from the protocol-paused state to the admin-paused state.

☐ Test coverage

- If `pool` is an invalid address, the `PoolStateLibrary._slot0` or `IMetricOmmPoolFactoryActions(pool).setPause` call fails.

☐ Test coverage

Function call analysis

- `PoolStateLibrary._slot0(pool)`
 - **What is controllable?** The `pool` address determines which contract is read. Reads slot 0 of the pool via `IExtload(pool).extload()`.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned `pauseLevel (uint8)` determines whether the state transition is allowed. If the pool is a malicious contract, it could return an arbitrary value to bypass the state machine check. However, since `setPause` is also called on the same pool in this scenario, the attacker's gain is limited.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if `extload` fails. Reentrancy is prevented by `nonReentrant`.
- `IMetricOmmPoolFactoryActions(pool).setPause(1)`
 - **What is controllable?** The `newLevel` value called (constant 1 here) and the target `pool` address are specified by the admin.
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value. An `InvalidPauseLevel` revert (when the level exceeds 2) could occur inside the pool, but since the constant 1 is passed here, it is not applicable. The pool-side `setPause` verifies `msg.sender == factory`, so direct calls from outside the factory are impossible.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the pool reverts, the entire transaction is rolled back. `nonReentrant` is

applied, preventing re-entry into the factory.

Security note: The pause state machine has the structure $0 \leftrightarrow 1$ (admin toggle), $0, 1 \rightarrow 2$ (protocol forced pause), and $2 \rightarrow 1$ (protocol downgrade). Since `pausePool` only allows the $0 \rightarrow 1$ transition, it is impossible for the admin to arbitrarily lower the level from the protocol-paused (level 2) state.

Function: `unpausePool(address pool)` external override `nonReentrant onlyPoolAdmin(pool)`

A function by which the pool admin lifts the admin-level pause and restores the pool to the active state (level 0). Only `poolAdmin[pool]` can call it, and the current `pauseLevel` must be exactly 1. The current state is read via `PoolStateLibrary._slot0`, and only the valid transition ($1 \rightarrow 0$) is allowed. Pools that have been force-paused by the protocol at level 2 cannot be unpaused via this function; `protocolUnpausePool` (factory owner only) must be used instead.

Inputs

- `pool`
 - **Control:** Pool admin-controlled
 - **Constraints:** Must be a pool registered in the factory, the caller must be the admin of that pool, and the current `pauseLevel` must be exactly 1.
 - **Impact:** Determines which pool's pause to lift.

Branches and code coverage

Intended branches

- When `pauseLevel == 1` and the caller is the pool admin, the pool's `pauseLevel` is changed to 0 and the `PauseLevelUpdated` event is emitted from the pool.

☒ Test coverage

Negative branches

- If `msg.sender` is not the pool admin, the function reverts with `NotPoolAdmin`.

☐ Test coverage

- If the pool is already active (`pauseLevel == 0`), the function reverts with `InvalidPauseTransition(0, 0)`.

☐ Test coverage

- If the pool is in protocol-paused state (`pauseLevel == 2`), the function reverts with

InvalidPauseTransition(2, 0), as the admin cannot lift the protocol's forced pause.

☒ Test coverage

- If pool is an invalid address, the PoolStateLibrary._slot0 or IMetricOmmPoolFactoryActions(pool).setPause call fails.

☐ Test coverage

Function call analysis

- PoolStateLibrary._slot0(pool)
 - **What is controllable?** The pool address determines which contract is read. Reads slot 0 of the pool via IExtstore(pool).extstore().
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned pauseLevel (uint8) determines whether the transition is allowed. Since pauseLevel is read directly from storage, the pool contract itself would have to be malicious in order to tamper with the read. For a legitimate pool, data falsification during the read is impossible.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if extstore fails. Reentrancy is prevented by nonReentrant.
- IMetricOmmPoolFactoryActions(pool).setPause(0)
 - **What is controllable?** The newLevel passed is the constant 0. The target pool address is specified by the admin.
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value. Since the pool internally verifies msg.sender == factory, a direct setPause(0) call from outside the factory is impossible. Since pause level 0 is a valid value on the pool side, an InvalidPauseLevel revert will not occur.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the pool reverts, the entire transaction is rolled back. nonReentrant is applied, preventing re-entry into the factory.

Security note: unpausePool and pausePool are symmetrically responsible only for the level 0<->1 transition. Level 2 (protocol pause) is a state the admin can never access, providing a strong security guarantee that the admin cannot arbitrarily resume a pool that the protocol owner has force-paused. Note that when pauseLevel == 2, the admin cannot even transition to level 1, meaning only the protocol owner can exit the level 2 state.

Function: `proposePoolPriceProvider(address pool, address newPriceProvider) external override nonReentrant onlyPoolAdmin(pool)`

This function registers a timelock proposal to replace the price provider of a pool. Only `poolAdmin[pool]` may call it; if the pool was created with an immutable price provider, it reverts with `PriceProviderImmutable`. When the proposal is accepted, `pendingPriceProvider[pool]` and `pendingPriceProviderExecuteAfter[pool]` are set, and the actual replacement can only occur after the timelock (`priceProviderTimelock[pool]`) has elapsed by calling `executePoolPriceProviderUpdate`. The new price provider is immediately validated via `_validatePriceProvider`.

Inputs

- `pool`
 - **Control:** Pool admin-controlled
 - **Constraints:** The caller must be `poolAdmin[pool]`. The pool must not have been created with an immutable price provider (condition: `p.immutablePriceProvider == address(0)`).
 - **Impact:** Determines which pool's price provider replacement proposal is registered.
- `newPriceProvider`
 - **Control:** Pool admin-controlled
 - **Constraints:** `_validatePriceProvider` checks whether the address is `address(0)` (`InvalidPriceProvider`) and calls `IPriceProvider(newPriceProvider).getTokens()` to verify that (`token0`, `token1`) matches the pool's tokens (`PriceProviderTokenMismatch`).
 - **Impact:** Stored in `pendingPriceProvider[pool]`; applied as the pool's actual price provider when `executePoolPriceProviderUpdate` is called after the timelock elapses. If an invalid price provider is applied, the pool's swap price calculation becomes corrupted.

Branches and code coverage

Intended branches

- When the pool has a mutable price provider and `newPriceProvider` passes `_validatePriceProvider`, `pendingPriceProvider[pool]` and `pendingPriceProviderExecuteAfter[pool]` are set and a `PoolPriceProviderChangeProposed` event is emitted.

☑ Test coverage

- When `PoolStateLibrary._slot3(pool)` returns `address(0)` (i.e., the mutable provider has not yet been written to `slot3`), current falls back to `p.immutablePriceProvider`. However, for a mutable pool `p.immutablePriceProvider` is `address(0)`, so current

also becomes `address(0)`.

☐ Test coverage

Negative branches

- If `msg.sender != poolAdmin[pool]`, the function reverts with `NotPoolAdmin`.

☐ Test coverage

- If `p.immutablePriceProvider != address(0)` (immutable price provider pool), the function reverts with `PriceProviderImmutable`.

☒ Test coverage

- If `newPriceProvider == address(0)`, the function reverts with `InvalidPriceProvider`.

☐ Test coverage

- If the token pair returned by `IPriceProvider(newPriceProvider).getTokens()` does not match the pool's tokens, the function reverts with `PriceProviderTokenMismatch`.

☐ Test coverage

- If `newPriceProvider` is a contract that does not implement `getTokens()`, the external call reverts.

☐ Test coverage

Function call analysis

- `IMetricOmmPool(pool).getImmutables()`

- **What is controllable?** The `pool` address determines the call target.
- **If the return value is controllable, how is it used and how can it go wrong?** `p.immutablePriceProvider`, `p.token0`, and `p.token1` from the returned `PoolImmutables` struct are used. In a legitimate pool contract these values are fixed at deployment and cannot be tampered with. A malicious `pool` address could return arbitrary values, but the `onlyPoolAdmin` check ensures only factory-registered addresses are valid.
- **What happens if it reverts, reenters, or does other unusual control flow?** A revert rolls back the entire transaction. Reentrancy is prevented by `nonReentrant`.

- `PoolStateLibrary._slot3(pool)`

- **What is controllable?** The `pool` address determines the read target.
- **If the return value is controllable, how is it used and how can it go wrong?** The returned `mutableProvider` address is only used as the current value in the event. It does not affect state-change logic, so the only consequence of an inaccurate value is incorrect event information.
- **What happens if it reverts, reenters, or does other unusual control flow?**

Reverts if extsload fails.

Security note: If the timelock (`priceProviderTimelock[pool]`) is `type(uint256).max`, the pool is treated as having an immutable price provider and this function itself reverts. If the timelock is 0, the proposal can be executed immediately in the same transaction, making it important to configure an appropriate timelock at pool creation.

Function: `executePoolPriceProviderUpdate(address pool)` external override nonReentrant onlyPoolAdmin(pool)

This function applies the price provider replacement proposed via `proposePoolPriceProvider` after the timelock has elapsed. Only `poolAdmin[pool]` may call it; if there is no pending proposal or the timelock has not yet elapsed, it reverts. Before applying the change, it re-validates the new price provider via `_validatePriceProvider`. On success, it deletes the `pendingPriceProvider` and `pendingPriceProviderExecuteAfter` mappings and reflects the change in the pool directly via `IMetricOmmPoolFactoryActions(pool).setPriceProvider`.

Inputs

- `pool`
 - **Control:** Pool admin-controlled
 - **Constraints:** The caller must be `poolAdmin[pool]`. `pendingPriceProvider[pool]` must not be `address(0)`, and `block.timestamp >= pendingPriceProviderExecuteAfter[pool]` must hold. Additionally, the pool must not have been created with an immutable price provider.
 - **Impact:** Determines which pool's price provider is replaced.

Branches and code coverage

Intended branches

- When the timelock has elapsed, `pendingPriceProvider[pool]` is valid, and re-validation passes, the pool's price provider is updated and the factory's pending state is cleared. A `PoolPriceProviderUpdated` event is emitted.
 - ☒ Test coverage
- For a pool with a timelock of 0, execution is possible immediately after `proposePoolPriceProvider`.
 - ☐ Test coverage

Negative branches

- If `msg.sender != poolAdmin[pool]`, the function reverts with `NotPoolAdmin`.
 - ☐ Test coverage
- If `pendingPriceProvider[pool] == address(0)` (no pending proposal), the function reverts with `NoPriceProviderChangeProposed`.
 - ☐ Test coverage
- If `block.timestamp < pendingPriceProviderExecuteAfter[pool]`, the function reverts with `PriceProviderTimelockNotElapsed`.
 - ☒ Test coverage
- If `p.immutablePriceProvider != address(0)` (immutable provider pool), the function reverts with `PriceProviderImmutable`; in theory this state should already be blocked by `proposePoolPriceProvider`, but the defensive check is applied redundantly.
 - ☐ Test coverage
- If the pending address has become invalid between proposal and execution (e.g., due to a token pair change or contract self-destruct), re-validation via `_validatePriceProvider` causes a revert.
 - ☐ Test coverage

Function call analysis

- `IMetricOmmPool(pool).getImmutables()`
 - **What is controllable?** The pool address determines the call target.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned `p.immutablePriceProvider`, `p.token0`, and `p.token1` are used for the immutable provider check and re-validation. In a legitimate pool these values are immutable and cannot be manipulated.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A revert rolls back the entire transaction. Reentrancy is prevented by `nonReentrant`.
- `IMetricOmmPoolFactoryActions(pool).setPriceProvider(pending)`
 - **What is controllable?** The pending address is set as the new price provider in the pool. The pool address is specified by the admin.
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value. Inside the pool, after verifying `msg.sender == factory`, the pending address is stored in mutable slot3 and a `PriceProviderUpdated` event is emitted. From this point on, all pool swaps use the new price provider.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the pool reverts, `delete pendingPriceProvider[pool]` and `delete pendingPriceProviderExecuteAfter[pool]` are not executed, leaving the pending state intact. The admin can attempt execution again. `nonReentrant`

is applied, preventing re-entry into the factory.

Security note: Pools with a timelock of 0 allow proposal and execution within the same block. Additionally, since `_validatePriceProvider` is called again at execution time, the scenario where `getTokens()` return values differ between proposal and execution must be considered for upgradeable `IPriceProvider` contracts. The ordering in which the pending state is deleted after the `setPriceProvider` call partially reverses the CEI (checks-effects-interactions) pattern, but `nonReentrant` compensates for this.

Function: `proposePoolAdminTransfer(address pool, address newAdmin)`

This is the first step of a two-step process for transferring admin rights of a specific pool to a new address. Only the current `poolAdmin[pool]` may call it (via the `onlyPoolAdmin` modifier), and it stores `newAdmin` in `pendingPoolAdmin[pool]`. The actual transfer of authority is not complete until the new admin calls `acceptPoolAdmin`. Transfers to the zero address or to the current admin address revert immediately.

Inputs

- `pool`
 - **Control:** Pool admin-controlled
 - **Constraints:** Must be a pool address registered in the factory, and `poolAdmin[pool]` must match `msg.sender` (passing an unregistered arbitrary address makes `poolAdmin[pool]` equal to `address(0)`, causing a `NotPoolAdmin` revert).
 - **Impact:** Determines which pool's pending admin slot is modified.
- `newAdmin`
 - **Control:** Pool admin-controlled
 - **Constraints:** `address(0)` is not allowed (`InvalidAdmin` revert); the same address as the current `poolAdmin[pool]` is not allowed (`InvalidAdmin` revert).
 - **Impact:** Stored in `pendingPoolAdmin[pool]` as the target for a subsequent `acceptPoolAdmin` call.

Branches and code coverage

Intended branches

- When `proposePoolAdminTransfer` is called successfully with a valid `newAdmin`, `pendingPoolAdmin[pool]` is set and a `PoolAdminTransferProposed` event is emitted.

☒ Test coverage

Negative branches

- If `msg.sender` is not `poolAdmin[pool]`, the function reverts with `NotPoolAdmin`.

☒ Test coverage

- If `newAdmin == address(0)`, the function reverts with `InvalidAdmin`.

☒ Test coverage

- If `newAdmin` equals the current pool admin, the function reverts with `InvalidAdmin`.

☒ Test coverage

- If `pendingPoolAdmin[pool]` is already set, it is overwritten without a revert, as there is no duplicate-proposal prevention logic.

☐ Test coverage

Function: `acceptPoolAdmin(address pool)`

This is the second step of the two-step pool admin transfer process. Only the address stored in `pendingPoolAdmin[pool]` may call this function to finalize the acquisition of admin rights. On success, `poolAdmin[pool]` is updated to `msg.sender` and `pendingPoolAdmin[pool]` is deleted. There is no `onlyPoolAdmin` restriction, so anyone may call it, but if `msg.sender != pending` the call reverts with `NotPendingPoolAdmin`. The `nonReentrant` modifier is applied.

Inputs

- `pool`
 - **Control:** Arbitrary
 - **Constraints:** `pendingPoolAdmin[pool]` must not be `address(0)` (`NoPendingPoolAdminTransfer revert`); `msg.sender == pendingPoolAdmin[pool]` must hold (`NotPendingPoolAdmin revert`).
 - **Impact:** Determines which pool's admin rights are accepted. Passing an invalid address (unregistered pool) makes `pendingPoolAdmin[pool]` equal to `address(0)`, causing a `NoPendingPoolAdminTransfer revert`.

Branches and code coverage

Intended branches

- When `pendingPoolAdmin[pool]` is set and `msg.sender == pending`, `poolAdmin[pool]` is updated, `pendingPoolAdmin[pool]` is deleted, and a `PoolAdminTransferred` event is emitted.

☒ Test coverage

Negative branches

- If `pendingPoolAdmin[pool] == address(0)` (no pending transfer), the function reverts with `NoPendingPoolAdminTransfer`.

☒ Test coverage

- If `msg.sender != pendingPoolAdmin[pool]`, the function reverts with `NotPendingPoolAdmin(pool, msg.sender, pending)`.

☒ Test coverage

- If a reentrancy attempt is made (e.g., a callback triggered after `poolAdmin[pool]` is updated), the `nonReentrant` modifier blocks it.

☐ Test coverage

Function call analysis

Function: `cancelPoolAdminTransfer(address pool)`

This function allows the current admin to cancel an in-progress pool admin transfer proposal. Only the current `poolAdmin[pool]` may call it (via the `onlyPoolAdmin` modifier), and it deletes `pendingPoolAdmin[pool]`. If there is no pending transfer (`pendingPoolAdmin[pool] == address(0)`), it reverts with `NoPendingPoolAdminTransfer`. The `nonReentrant` modifier is applied.

Inputs

- `pool`
 - **Control:** Pool admin-controlled
 - **Constraints:** `poolAdmin[pool]` must match `msg.sender`; `pendingPoolAdmin[pool]` must not be `address(0)`.
 - **Impact:** Determines which pool's pending admin transfer is cancelled.

Branches and code coverage

Intended branches

- When `pendingPoolAdmin[pool]` is set and `msg.sender == poolAdmin[pool]`, `pendingPoolAdmin[pool]` is deleted and a `PoolAdminTransferCancelled` event is emitted.

☒ Test coverage

Negative branches

- If `msg.sender` is not `poolAdmin[pool]`, the function reverts with `NotPoolAdmin`.
 - ☐ Test coverage
- If `pendingPoolAdmin[pool] == address(0)` (no transfer to cancel), the function reverts with `NoPendingPoolAdminTransfer`.
 - ☐ Test coverage
- When an unregistered pool address is passed, `poolAdmin[pool]` is `address(0)`, causing the function to revert with `NotPoolAdmin`.
 - ☐ Test coverage

Function: `addLiquidityExactShares(address pool, address owner, uint80 salt, LiquidityDelta calldata deltas, uint256 maxAmountToken0, uint256 maxAmountToken1, bytes calldata hookData)` external override returns `(uint256 amount0Added, uint256 amount1Added)`

`addLiquidityExactShares` is the entry point for adding liquidity to a pool on behalf of an EOA by specifying per-bin shares explicitly. After validating `owner` and `deltas`, it enters `_addLiquidity`, which stores a transient pay context (`pool`, `payer=msg.sender`, `max0`, `max1`) and calls `IMetricOmmPoolActions(pool).addLiquidity(owner, salt, deltas, abi.encode(KIND_PAY), hookData)` within a `try/catch`. The pool calls back into `metricOmmModifyLiquidityCallback` during execution to pull tokens from the payer, and if the required amount exceeds the `max0/max1` ceiling, the function reverts with `MaxAmountExceeded`. An overload that omits `owner` and uses `msg.sender` as the owner exists, but the core logic is identical.

Inputs

- `pool`
 - **Control:** Arbitrary
 - **Constraints:** No whitelist. Settlement is established via the `msg.sender == expectedPool` check in the callback.
 - **Impact:** The pool to add liquidity to.
- `owner`
 - **Control:** Arbitrary
 - **Constraints:** `_validateOwner` reverts with `InvalidPositionOwner` if `address(0)`.
 - **Impact:** The position owner recorded by the pool. Can be different from the payer (`msg.sender`), allowing a third party to fund another person's position.
- `salt`

- **Control:** Arbitrary
 - **Constraints:** No validation.
 - **Impact:** A key that distinguishes positions belonging to the same owner.
- `deltas`
 - **Control:** Arbitrary
 - **Constraints:** `_validateDeltas` reverts with `LiquidityDeltaLengthMismatch` if `binIdxs.length != shares.length`, and with `EmptyLiquidityDelta` if the length is 0.
 - **Impact:** Per-bin shares to add. The pool converts these into token requirements.
- `maxAmountToken0 / maxAmountToken1`
 - **Control:** Arbitrary
 - **Constraints:** If `amount0Delta > max0 || amount1Delta > max1` in the callback, reverts with `MaxAmountExceeded`. The comparison is inclusive (`>`).
 - **Impact:** Slippage ceiling. Blocks settlement if token requirements increase due to price movement.
- `hookData`
 - **Control:** Arbitrary
 - **Constraints:** No validation. Passed to the pool/hook.
 - **Impact:** Input to hook logic.

Branches and code coverage

Intended branches

- When `owner != address(0)`, `deltas` is valid, and the callback requirements are within the ceiling, liquidity is added normally.
 - ☑ Test coverage
- When taking the overload path (`owner == msg.sender`), the function behaves identically.
 - ☑ Test coverage
- The payer is always fixed as `msg.sender`, so only the caller's own funds are pulled in the callback. Since `owner` can be specified arbitrarily, contributing funds to a third party's position is supported.
 - ☑ Test coverage
- External calls go through the internal `_addLiquidity`. Before calling `pool.addLiquidity`, `_addLiquidity` sets the transient pay context via `_setPayContext(pool, payer=msg.sender, max0, max1)` (reverting with `PayContextAlreadyActive` if already active), and clears the context via `_clearPayContext` on both success and failure paths. The callback

(metricOmmModifyLiquidityCallback) loads this context, validates the caller and ceiling, then performs settlement.

☒ Test coverage

Negative branches

- If `owner == address(0)`, the function reverts with `InvalidPositionOwner`.

☒ Test coverage

- If `deltas.binIdxs.length != deltas.shares.length`, the function reverts with `LiquidityDeltaLengthMismatch`.

☐ Test coverage

- If `deltas.binIdxs.length == 0`, the function reverts with `EmptyLiquidityDelta`.

☒ Test coverage

- If the pay context is already active, `_setPayContext` reverts with `PayContextAlreadyActive`.

☐ Test coverage

- If the callback's required amount exceeds `max0/max1`, the function reverts with `MaxAmountExceeded(...)`.

☒ Test coverage

- If `pool.addLiquidity` reverts, the catch block clears the pay context and bubbles up the original reason.

☐ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).addLiquidity(owner, salt, deltas, abi.encode(KIND_PAY), hookData)` (via `internal_addLiquidity`)

- **What is controllable?** `pool`, `owner`, `salt`, `deltas`, and `hookData` are caller-controlled. `callbackData` is fixed as `KIND_PAY`.
- **If the return value is controllable, how is it used and how can it go wrong?** The returned `(amount0Added, amount1Added)` is passed directly to the caller. Actual fund protection is provided by the `max0/max1` ceiling check in the callback.
- **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped in `try/catch`; on revert, the pay context is cleared and the original reason is bubbled up. The pool calls back into the callback; `_setPayContext` prevents duplicate activation with `PayContextAlreadyActive`, guarding against settlement errors due to reentrancy.

Function: `addLiquidityWeighted(address pool, address owner, uint80 salt, LiquidityDelta calldata weightDeltas, uint256 maxAmountToken0, uint256 maxAmountToken1, bytes calldata hookData)` **external override**
returns `(uint256 amount0Added, uint256 amount1Added)`

`addLiquidityWeighted` accepts a weight vector (`weightDeltas`) instead of shares, first performs a **probe** call using the weights as provisional shares to determine the required token amounts (`need0/need1`), then rescales the shares by the ratio `min(max0/need0, max1/need1)` and executes the actual addition. The probe calls `pool.addLiquidity` with `KIND_PROBE` `callbackData`, and since the callback always reverts with `LiquidityProbe(need0, need1)`, the pool state is not modified. The probe revert is caught and decoded within the `try/catch` block, and the `_addLiquidity` (`KIND_PAY`) call is then executed with the shares scaled to integers. An overload that omits `owner` exists, but the core logic is identical.

Inputs

- `pool`
 - **Control:** Arbitrary
 - **Constraints:** No whitelist.
 - **Impact:** The pool to add liquidity to.
- `owner`
 - **Control:** Arbitrary
 - **Constraints:** `_validateOwner` reverts with `InvalidPositionOwner` if `address(0)`.
 - **Impact:** Position owner.
- `salt`
 - **Control:** Arbitrary
 - **Constraints:** No validation.
 - **Impact:** Position distinction key.
- `weightDeltas`
 - **Control:** Arbitrary
 - **Constraints:** Subject to `_validateDeltas` (length match/non-zero) and `_validatePositiveWeights`, which reverts with `ZeroWeight` if any share is 0.
 - **Impact:** Used as provisional shares that serve as the basis for the probe and scale calculation.
- `maxAmountToken0 / maxAmountToken1`
 - **Control:** Arbitrary
 - **Constraints:** Used as scale numerators. Also re-validated as a ceiling in the callback.
 - **Impact:** The smaller of the two token ceiling ratios determines the final scale.

- hookData
 - **Control:** Arbitrary
 - **Constraints:** No validation.
 - **Impact:** Input to hook logic.

Branches and code coverage

Intended branches

- When the probe reverts with `LiquidityProbe`, the catch block decodes `need0/need1` and executes the scaled addition.
 - ☒ Test coverage
- When `need0 == 0` or `need1 == 0`, the corresponding leg is treated as unconstrained (`type(uint256).max`) and only the other leg determines the scale.
 - ☐ Test coverage
- When both legs are positive, the scale `min(max0/need0, max1/need1)` is applied.
 - ☒ Test coverage
- Since the probe and actual addition are separate `pool.addLiquidity` calls, price or oracle state may change between the two calls, potentially making `need` stale; however, the `max0/max1` ceiling re-validation in the callback prevents overspending.
 - ☐ Test coverage
- The probe call is performed **directly** within `addLiquidityWeighted` without setting a pay context (since the callback reverts immediately under `KIND_PROBE` without caller validation, resulting in no state changes or settlement). Only the actual addition goes through the internal `_addLiquidity` with the `_setPayContext/_clearPayContext` lifecycle, during which the callback validates the caller and ceiling before settling.
 - ☐ Test coverage

Negative branches

- If `owner == address(0)`, the function reverts with `InvalidPositionOwner`.
 - ☐ Test coverage
- If `deltas` length mismatches or is 0, the function reverts with `LiquidityDeltaLengthMismatch` or `EmptyLiquidityDelta`.
 - ☐ Test coverage
- If any `weightDeltas.shares[i] == 0`, the function reverts with `ZeroWeight`.
 - ☒ Test coverage
- If the probe returns normally instead of reverting with `LiquidityProbe`, the function reverts with `WeightedProbeInconclusive`.

- ☐ Test coverage
 - If the probe revert reason is shorter than 68 bytes, the function reverts with `UnexpectedRevertLength(len)` (via `_decodeLiquidityProbeOrBubble`).
- ☐ Test coverage
 - If the probe revert selector is not `LiquidityProbe.selector`, the original reason is bubbled up.
- ☐ Test coverage
 - If the scale result rounds any positive weight down to share 0, the function reverts with `SharesRoundedToZero`.
- ☐ Test coverage
 - If the actual addition (`_addLiquidity`) exceeds the callback ceiling, the function reverts with `MaxAmountExceeded`.
- ☐ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).addLiquidity(owner, salt, weightDeltas, abi.encode(KIND_PROBE), hookData) (probe)`
 - **What is controllable?** `pool`, `owner`, `salt`, `weightDeltas`, and `hookData`. `callbackData` is fixed as `KIND_PROBE`.
 - **If the return value is controllable, how is it used and how can it go wrong?** A normal return is an unintended path and is handled with `WeightedProbeInconclusive`. In the normal flow, the callback reverts with `LiquidityProbe` to pass `need0/need1` back as a return channel.
 - **What happens if it reverts, reenters, or does other unusual control flow?** The revert reason is decoded via inline assembly. If the selector or length does not match expectations, it is safely bubbled up or blocked with `UnexpectedRevertLength`.
- `IMetricOmmPoolActions(pool).addLiquidity(owner, salt, scaled, abi.encode(KIND_PAY), hookData) (actual addition, via _addLiquidity)`
 - **What is controllable?** `scaled` is an internally computed result; the rest are caller-controlled.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned added amounts are passed directly to the caller. Fund protection is provided by the callback ceiling check.
 - **What happens if it reverts, reenters, or does other unusual control flow?** `_addLiquidity`'s `try/catch` clears the pay context and bubbles up the reason.

Function: `metricOmmModifyLiquidityCallback(uint256 amount0Delta, uint256 amount1Delta, bytes calldata callbackData)` **external override**

`metricOmmModifyLiquidityCallback` is the callback invoked by the pool during `addLiquidity` execution for settlement, distinguishing between probe and pay modes based on the `kind` field in `callbackData`. If `KIND_PROBE`, it immediately reverts with `LiquidityProbe(amount0Delta, amount1Delta)` to return the required token amounts to the caller without modifying state. If `KIND_PAY`, it loads the transient pay context, validates the caller and ceiling, then transfers `token0`/`token1` from the payer to the pool via `safeTransferFrom`.

Inputs

- `amount0Delta`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Computed and passed by the pool. If `> max0` in `KIND_PAY`, reverts with `MaxAmountExceeded`.
 - **Impact:** `token0` requirement (a reported value in probe; the transfer amount in pay).
- `amount1Delta`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Computed and passed by the pool. If `> max1` in `KIND_PAY`, reverts with `MaxAmountExceeded`.
 - **Impact:** `token1` requirement.
- `callbackData`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** `kind` is obtained via `abi.decode(_, (uint8))`. Values other than `KIND_PROBE(0)` and `KIND_PAY(1)` result in `InvalidCallbackKind`.
 - **Impact:** Determines the settlement mode. Since the adder encodes this directly when calling the pool, it is difficult to set arbitrarily from outside.

Branches and code coverage

Intended branches

- When `kind == KIND_PROBE`, the function reverts with `LiquidityProbe(amount0Delta, amount1Delta)` as the return channel in the normal flow.
 - ☑ Test coverage
- When `kind == KIND_PAY`, the function validates the context and ceiling, then transfers tokens for positive deltas.
 - ☑ Test coverage
- When `amount0Delta > 0`, `safeTransferFrom` is called for `token0`; when `amount1Delta`

> 0, safeTransferFrom is called for token1.

☐ Test coverage

- Since KIND_PROBE reverts unconditionally without caller validation, it has no side effects. Regardless of who calls this selector, the function simply returns the input values as an error without modifying state.

☒ Test coverage

- The payer is the original caller stored in the pay context, so the callback cannot pull tokens from an arbitrary address. After successful transfer, the context is cleared with _clearPayContext.

☒ Test coverage

Negative branches

- If kind is neither PROBE nor PAY, the function reverts with InvalidCallbackKind.

☐ Test coverage

- If kind == KIND_PAY but the pay context is not active (expectedPool == address(0)), the function reverts with CallbackContextNotActive.

☐ Test coverage

- If msg.sender != expectedPool, the function reverts with InvalidCallbackCaller(msg.sender, expectedPool).

☐ Test coverage

- If amount0Delta > max0 || amount1Delta > max1, the function reverts with MaxAmountExceeded(amount0Delta, amount1Delta, max0, max1).

☒ Test coverage

Function call analysis

- IMetricOmmPool(msg.sender).getImmutables()
 - **What is controllable?** msg.sender is the pool that has been validated to match expectedPool.
 - **If the return value is controllable, how is it used and how can it go wrong?** The token0/token1 addresses determine the tokens to transfer. Only a pool that passes caller validation can reach this point.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If this view call reverts, the callback fails and the entire addLiquidity reverts.
- IERC20(token0/token1).safeTransferFrom(payer, msg.sender, amountDelta)
 - **What is controllable?** payer is the original caller fixed in the context; the amount is a pool-computed value that has passed the ceiling check.
 - **If the return value is controllable, how is it used and how can it go wrong?**

Succeeds only within the limit pre-approved by the payer on the adder.
Fee-on-transfer tokens may result in the pool receiving less than expected, potentially causing validation failures on the pool side.

- **What happens if it reverts, reenters, or does other unusual control flow?** A transfer failure causes the callback to revert, rolling back the entire addition.

Function: swap(address pool, address recipient, bool zeroForOne, int128 amountSpecified, uint128 priceLimitX64, uint256 deadline) public payable returns (int128 amount0Delta, int128 amount1Delta)

swap is the base swap entry point of the swapper. It executes a single swap against an arbitrary pool using the caller as the payer, and settles the input token via a callback. Internally, it checks deadline and enforces `msg.value == 0` (not a native path), then delegates to `_swapWithContext`. `_swapWithContext` validates direction-specific sentinels via `_validatePriceLimit` and sets a transient swap context (payer/pool/flags) via `_startSwap` to prevent concurrent swaps, then calls `IMetricOmmPoolActions(pool).swap(...)` within a try/catch. The pool calls back into `metricOmmSwapCallback` during the swap to settle the input token, and the returned deltas are validated by `_validateAmountSpecifiedMatch` to ensure the sign and magnitude match `amountSpecified`. Two overloads that accept data and hookData exist, but the core logic is identical (here the bytes payload is empty).

Inputs

- pool
 - **Control:** Arbitrary
 - **Constraints:** No whitelist validation is performed. The caller-specified address is used as the pool, and settlement proceeds only if the swap context's pool matches the callback's `msg.sender`.
 - **Impact:** Any contract can be specified as the pool, but settlement will fail unless it is a valid pool that passes callback caller validation and delta verification.
- recipient
 - **Control:** Arbitrary
 - **Constraints:** No `address(0)` validation. The address that receives the output token, passed directly to the pool.
 - **Impact:** If specified incorrectly, the output token is sent to an unrecoverable address.
- zeroForOne
 - **Control:** Arbitrary
 - **Constraints:** `true` means token0→token1 (price decreases); `false` means the

- opposite direction. Determines which sentinel is permitted by `_validatePriceLimit`.
 - **Impact:** Determines the swap direction, the input token to settle in the callback, and the sign validation branch for the resulting delta.
- `amountSpecified`
 - **Control:** Arbitrary
 - **Constraints:** `int128`. Positive means `exactIn`; negative means `exactOut`. A value of 0 is rejected by the pool with `InvalidAmount`.
 - **Impact:** The signed amount specification passed to the pool, determining `exactIn/exactOut` mode. The returned delta is validated against this value for sign and range by `_validateAmountSpecifiedMatch`.
- `priceLimitX64`
 - **Control:** Arbitrary
 - **Constraints:** If `zeroForOne == true`, `type(uint128).max` is forbidden (`InvalidPriceLimitForDirection`) and 0 is accepted as the no-lower-bound sentinel. If `zeroForOne == false`, 0 is forbidden and `type(uint128).max` is accepted as the no-upper-bound sentinel.
 - **Impact:** Passed to the pool as the price limit for the swap. Using a sentinel allows traversal of all bins without a price guard.
- `deadline`
 - **Control:** Arbitrary
 - **Constraints:** If `block.timestamp > deadline`, reverts with `TransactionExpired`. `deadline == block.timestamp` is permitted.
 - **Impact:** Sets the upper time bound for the transaction, preventing stale execution due to mempool delays.
- `data / hookData` (overloads only)
 - **Control:** Arbitrary
 - **Constraints:** No length limit or validation. In the base overload, empty values (`""`, `hex""`) are used.
 - **Impact:** Passed as `callbackData/hookData` to the pool for use in callback and hook logic.

Branches and code coverage

Intended branches

- When `amountSpecified > 0` (`exactIn`) or `amountSpecified < 0` (`exactOut`), the respective branch executes normally.
- ☒ Test coverage
 - When `zeroForOne == true` or `false`, the respective swap direction is taken.

☒ Test coverage

- When `priceLimitX64` is the direction-appropriate sentinel (0 for `zeroForOne`, `type(uint128).max` otherwise), the swap proceeds without a price guard.

☒ Test coverage

- After `pool.swap` succeeds, `_validateAmountSpecifiedMatch` passes because the returned delta matches the specification.

☒ Test coverage

- The swap context is intentionally retained after `pool.swap` succeeds so it can be used during callback settlement; the entry function (`_swap`) clears it with `_clearSwap` at the end. In the catch path, the context is explicitly cleared before the reason is bubbled up.

☒ Test coverage

Negative branches

- If `block.timestamp` exceeds deadline, the function reverts with `TransactionExpired(deadline, block.timestamp)`.

☒ Test coverage

- If `msg.value != 0`, the function reverts with `NativeValueNotExpected` (this path does not accept native input).

☐ Test coverage

- If `zeroForOne == true` and `priceLimitX64 == type(uint128).max`, the function reverts with `InvalidPriceLimitForDirection(true, ...)`.

☒ Test coverage

- If `zeroForOne == false` and `priceLimitX64 == 0`, the function reverts with `InvalidPriceLimitForDirection(false, ...)`.

☒ Test coverage

- If a swap context is already active (`currentPool != address(0)`), reentering reverts with `SwapInProgress`.

☒ Test coverage

- If `IMetricOmmPoolActions(pool).swap` reverts, the catch block clears the swap context and bubbles up the original reason as-is.

☐ Test coverage

- If the returned delta mismatches the sign or range of `amountSpecified`, the function reverts with `AmountSpecifiedMismatch(amountSpecified, actual)`.

☐ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).swap(recipient, zeroForOne, amountSpecified, priceLimitX64, callbackData, hex"")`
 - **What is controllable?** `pool`, `recipient`, `zeroForOne`, `amountSpecified`, `priceLimitX64`, and `callbackData` are all caller-controlled. `hookData` is empty in the base overload.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned `(amount0Delta, amount1Delta)` is validated by `_validateAmountSpecifiedMatch` before being returned to the caller. For `exactIn`, the actual input delta must be within `(0, amountSpecified]`; for `exactOut`, the output delta must meet or exceed the specification. Abnormal values returned by the pool are blocked by `AmountSpecifiedMismatch`.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped in `try/catch`; on revert, the swap context is cleared and the original reason is bubbled up via inline assembly. During the swap, the pool calls back into `metricOmmSwapCallback`. Since there is no separate `nonReentrant` and only the transient swap context provides protection, initiating a new swap on the same swapper from within the callback reverts with `SwapInProgress`.
- `metricOmmSwapCallback(amount0Delta, amount1Delta, data)` (settlement callback called back by the pool)
 - **What is controllable?** Called by the pool; the swapper verifies that `msg.sender == context.pool`. Delta values are computed by the pool.
 - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value. The swapper transfers the input token from the payer to the pool via `safeTransferFrom` for positive deltas. The payer is fixed as `msg.sender` at swap entry, so the caller must have previously approved the swapper.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the callback transfer fails, the entire `pool.swap` call reverts and flows into the catch block. See the `metricOmmSwapCallback` documentation for detailed settlement flow.

Function: `swapExactInput(address pool, address recipient, bool zeroForOne, uint128 amountIn, uint128 priceLimitX64, uint256 minAmountOut, uint256 deadline)` external payable returns (uint256 amountOut, uint256 amountInUsed)

`swapExactInput` is the entry point for performing an ERC-20-to-ERC-20 swap with an exact input amount (`amountIn`) and a minimum output slippage guard (`minAmountOut`). It converts `amountIn` into a positive signed amount specification via `_toSignedExactInput` (reverting with

AmountTooLarge if it exceeds `int128.max`), then calls internal `_swap` with an empty payload. The returned deltas are decoded into `(amountInUsed, amountOut)` by `_decodeSwapResult`, and the function reverts if `amountOut < minAmountOut`. A `hookData` overload exists but the core logic is identical.

Inputs

- `pool`
 - **Control:** Arbitrary
 - **Constraints:** No whitelist. Only a valid pool can complete settlement via callback caller validation.
 - **Impact:** Specifies the pool to swap against.
- `recipient`
 - **Control:** Arbitrary
 - **Constraints:** No `address(0)` validation.
 - **Impact:** Address to receive the output token.
- `zeroForOne`
 - **Control:** Arbitrary
 - **Constraints:** Direction flag. Determines the sentinel range permitted by `_validatePriceLimit`.
 - **Impact:** Determines the swap direction and the input/output tokens.
- `amountIn`
 - **Control:** Arbitrary
 - **Constraints:** `uint128`. If `amountIn > type(int128).max`, `_toSignedExactInput` reverts with `AmountTooLarge(amountIn)`.
 - **Impact:** Converted to a positive `amountSpecified` and passed to the pool as the exact input amount.
- `priceLimitX64`
 - **Control:** Arbitrary
 - **Constraints:** Direction-specific sentinel rules are identical to swap.
 - **Impact:** Price limit for the swap.
- `minAmountOut`
 - **Control:** Arbitrary
 - **Constraints:** No upper bound. Used only as a slippage lower bound and does not affect `amountSpecified` calculation.
 - **Impact:** If `amountOut < minAmountOut`, the function reverts with `InsufficientOutput(amountOut, minAmountOut)` to enforce the minimum received amount.
- `deadline`

- **Control:** Arbitrary
- **Constraints:** If `block.timestamp > deadline` in the internal `_swap`, reverts with `TransactionExpired`.
- **Impact:** Upper time bound for the transaction.

Branches and code coverage

Intended branches

- When `zeroForOne == true` or `false`, the respective exactIn swap executes normally.
 - ☐ Test coverage
- When `amountOut >= minAmountOut`, the function returns normally.
 - ☐ Test coverage
- Internally, the execution follows the `_swap` → `_swapWithContext` path, which sequentially performs: `_validatePriceLimit` (direction-specific sentinel validation), `_startSwap` (transient swap context setup, blocking duplicate swaps with `SwapInProgress`), `pool.swap` invocation, and `_decodeSwapResult` decomposition of the returned deltas. The only additional validations unique to this function are the `int128.max` upper bound on `amountIn` and the `minAmountOut` slippage lower bound.
 - ☐ Test coverage
- During the swap, the pool calls back into `metricOmmSwapCallback` to settle the input token from the payer (`msg.sender`). Since the swapper has no separate `nonReentrant` and relies solely on the transient context, reentering the same swapper from within the callback reverts with `SwapInProgress`.
 - ☐ Test coverage

Negative branches

- If `msg.value != 0`, the function reverts with `NativeValueNotExpected`.
 - ☐ Test coverage
- If `amountIn > type(int128).max`, the function reverts with `AmountTooLarge(amountIn)`.
 - ☐ Test coverage
- If `block.timestamp > deadline`, the function reverts with `TransactionExpired` (within internal `_swap`).
 - ☐ Test coverage
- If the `priceLimitX64` sentinel does not match the direction, the function reverts with `InvalidPriceLimitForDirection`.
 - ☐ Test coverage
- If `_decodeSwapResult` encounters a delta with an unexpected sign (e.g., `amount0Delta`

`<= 0 || amount1Delta >= 0`), the function reverts with `InvalidSwapDeltas`.

☐ Test coverage

- If `amountOut < minAmountOut`, the function reverts with `InsufficientOutput(amountOut, minAmountOut)`.

☐ Test coverage

- If `pool.swap` reverts, the catch block clears the swap context and bubbles up the original reason.

☐ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).swap(recipient, zeroForOne, _toSignedExactInput(amountIn), priceLimitX64, "", "")` (via internal `_swap` → `_swapWithContext`)
 - **What is controllable?** `pool`, `recipient`, `zeroForOne`, `amountIn`, and `priceLimitX64` are caller-controlled. `callbackData` and `hookData` are fixed as empty values.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned `(amount0Delta, amount1Delta)` is decomposed by `_decodeSwapResult` into input and output amounts, then validated by the `minAmountOut` slippage guard. Deltas with abnormal signs are blocked by `InvalidSwapDeltas`.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped in `try/catch`; on revert, the swap context is cleared and the original reason is bubbled up. During the swap, the pool calls back into `metricOmmSwapCallback` to settle the input token (see the `metricOmmSwapCallback` documentation for details).

Function: `swapExactOutput(address pool, address recipient, bool zeroForOne, uint128 amountOutDesired, uint128 priceLimitX64, uint256 maxAmountIn, uint256 deadline)` external payable returns `(uint256 amountOut, uint256 amountInUsed)`

`swapExactOutput` is the entry point for performing an ERC-20-to-ERC-20 swap with a desired output amount (`amountOutDesired`) and a maximum input guard (`maxAmountIn`). It converts `amountOutDesired` into a negative amount specification via `_toSignedExactOutput` (reverting with `AmountTooLarge` if it exceeds `int128.max`), then calls internal `_swap`. After decomposing the returned deltas with `_decodeSwapResult`, it reverts with `InvalidSwapDeltas` if partial fill results in `amountOut < amountOutDesired`, and with `InputTooHigh` if `amountInUsed > maxAmountIn`. A `hookData` overload exists but the core logic is identical.

Inputs

- pool
 - **Control:** Arbitrary
 - **Constraints:** No whitelist.
 - **Impact:** The pool to swap against.
- recipient
 - **Control:** Arbitrary
 - **Constraints:** No address(0) validation.
 - **Impact:** Address to receive the output token.
- zeroForOne
 - **Control:** Arbitrary
 - **Constraints:** Direction flag.
 - **Impact:** Determines the swap direction and input/output tokens.
- amountOutDesired
 - **Control:** Arbitrary
 - **Constraints:** uint128. If amountOutDesired > type(int128).max, _toSignedExactOutput reverts with AmountTooLarge. Converted to a negative amountSpecified.
 - **Impact:** The desired exact output amount. The pool may partially fill based on available liquidity.
- priceLimitX64
 - **Control:** Arbitrary
 - **Constraints:** Direction-specific sentinel rules are the same.
 - **Impact:** Price limit for the swap.
- maxAmountIn
 - **Control:** Arbitrary
 - **Constraints:** Used only as a slippage upper bound and does not affect amountSpecified calculation.
 - **Impact:** If amountInUsed > maxAmountIn, the function reverts with InputTooHigh(amountInUsed, maxAmountIn) to prevent overspending.
- deadline
 - **Control:** Arbitrary
 - **Constraints:** If block.timestamp > deadline, reverts with TransactionExpired.
 - **Impact:** Upper time bound for the transaction.

Branches and code coverage

Intended branches

- When `zeroForOne == true` or `false`, the respective `exactOut` swap executes normally.
 - ☐ Test coverage
- When `amountOut >= amountOutDesired` and `amountInUsed <= maxAmountIn`, the function returns normally.
 - ☐ Test coverage
- Internally, the execution follows the `_swap → _swapWithContext` path, which sequentially performs: `_validatePriceLimit` (direction-specific sentinel validation), `_startSwap` (transient swap context setup, blocking duplicate swaps with `SwapInProgress`), `pool.swap` invocation, and `_decodeSwapResult` decomposition of the returned deltas. The only additional validations unique to this function are the `int128.max` upper bound on `amountOutDesired` and the `maxAmountIn` input ceiling.
 - ☐ Test coverage
- During the swap, the pool calls back into `metricOmmSwapCallback` to settle the input token from the payer (`msg.sender`); the transient context alone provides reentrancy protection, so reentering the same swapper from within the callback reverts with `SwapInProgress`.
 - ☐ Test coverage

Negative branches

- If `msg.value != 0`, the function reverts with `NativeValueNotExpected`.
 - ☐ Test coverage
- If `amountOutDesired > type(int128).max`, the function reverts with `AmountTooLarge`.
 - ☐ Test coverage
- If `block.timestamp > deadline`, the function reverts with `TransactionExpired`.
 - ☐ Test coverage
- If the `priceLimitX64` sentinel does not match the direction, the function reverts with `InvalidPriceLimitForDirection`.
 - ☐ Test coverage
- If `_decodeSwapResult` sign validation fails, the function reverts with `InvalidSwapDeltas`.
 - ☐ Test coverage
- If partial fill results in `amountOut < amountOutDesired`, the function reverts with `InvalidSwapDeltas`.
 - ☐ Test coverage
- If `amountInUsed > maxAmountIn`, the function reverts with
 - ☐ Test coverage

```
InputTooHigh(amountInUsed, maxAmountIn).
```

☐ Test coverage

- If the pool cannot fulfill the specified amount due to price limit or liquidity constraints in an exactOut swap, `amountOut < amountOutDesired` results in an explicit failure via `InvalidSwapDeltas`. Partial fills are therefore not treated as success.

☐ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).swap(recipient, zeroForOne, _toSignedExactOutput(amountOutDesired), priceLimitX64, "", "")` (via internal `_swap → _swapWithContext`)
 - **What is controllable?** `pool`, `recipient`, `zeroForOne`, `amountOutDesired`, and `priceLimitX64` are caller-controlled. `callbackData` and `hookData` are fixed as empty values.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned deltas are decomposed by `_decodeSwapResult` and validated for output sufficiency (`amountOut >= amountOutDesired`) and input ceiling (`amountInUsed <= maxAmountIn`). If the pool consumes more input than specified, `InputTooHigh` blocks the transaction.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped in `try/catch`; on revert, the swap context is cleared and the original reason is bubbled up. During the swap, the pool calls back into `metricOmmSwapCallback` to settle the input token (see the `metricOmmSwapCallback` documentation for details).

Function: `swapExactInputNativeForTokens(address pool, address recipient, bool zeroForOne, uint128 amountIn, uint128 priceLimitX64, uint256 minAmountOut, uint256 deadline)` external payable returns (uint256 amountOut, uint256 amountInUsed)

`swapExactInputNativeForTokens` is the exactIn entry point that accepts native ETH as input and swaps it for tokens. It requires `msg.value == amountIn` exactly, and calls `_swapWithContext` with `payerIsNative = true`. During callback settlement, the swapper wraps its ETH balance into WETH and transfers it to the pool (reverting with `NativeInputNotSupported` if the input token is not WETH). After the swap, it reverts if `amountOut < minAmountOut`, and refunds any unused native (`msg.value - amountInUsed`) to the caller.

Inputs

- pool
 - **Control:** Arbitrary
 - **Constraints:** No whitelist.
 - **Impact:** The pool to swap against.
- recipient
 - **Control:** Arbitrary
 - **Constraints:** No address(0) validation.
 - **Impact:** Address to receive the output token.
- zeroForOne
 - **Control:** Arbitrary
 - **Constraints:** Direction flag. The input token must be WETH for native wrap settlement to be possible.
 - **Impact:** Determines the swap direction and input token.
- amountIn
 - **Control:** Arbitrary
 - **Constraints:** msg.value == uint256(amountIn) is required (otherwise reverts with InsufficientNativeValue). If it exceeds int128.max in _toSignedExactInput, reverts with AmountTooLarge.
 - **Impact:** The exact native input amount.
- priceLimitX64
 - **Control:** Arbitrary
 - **Constraints:** Direction-specific sentinel rules are the same.
 - **Impact:** Price limit for the swap.
- minAmountOut
 - **Control:** Arbitrary
 - **Constraints:** Slippage lower bound.
 - **Impact:** If amountOut < minAmountOut, the function reverts with InsufficientOutput.
- deadline
 - **Control:** Arbitrary
 - **Constraints:** If block.timestamp > deadline, reverts with TransactionExpired.
 - **Impact:** Upper time bound for the transaction.

Branches and code coverage

Intended branches

- When `msg.value == amountIn`, the normal native input swap executes.
 - ☒ Test coverage
- When `amountInUsed < msg.value`, the unused native amount is refunded to the caller.
 - ☐ Test coverage
- When `amountInUsed == msg.value`, the refund amount is 0 and `_refundUnusedNative` returns early.
 - ☒ Test coverage
- Although `_refundUnusedNative` also checks `InputTooHigh` when `amountInUsed > maxAmountIn` (which is `msg.value` here), in the normal path `amountInUsed <= msg.value` holds, so only the refund difference is transferred.
 - ☐ Test coverage
- Internally, `_swapWithContext` is called with `payerIsNative=true` and `expectNativeOutput=false`. It performs price limit sentinel validation, transient swap context setup (`SwapInProgress`), `pool.swap` invocation, and `_decodeSwapResult` decomposition, followed by the `minAmountOut` slippage guard. During callback settlement, the swapper wraps its ETH into WETH and transfers it to the pool (see the `metricOmmSwapCallback` documentation). After the swap, unused native is refunded via `_refundUnusedNative`.
 - ☒ Test coverage

Negative branches

- If `block.timestamp > deadline`, the function reverts with `TransactionExpired`.
 - ☐ Test coverage
- If `msg.value != amountIn`, the function reverts with `InsufficientNativeValue(amountIn, msg.value)`.
 - ☐ Test coverage
- If the `priceLimitX64` sentinel does not match the direction, the function reverts with `InvalidPriceLimitForDirection`.
 - ☐ Test coverage
- If the input token is not WETH (within callback `_payInput`), the function reverts with `NativeInputNotSupported(token)`.
 - ☐ Test coverage
- If `amountOut < minAmountOut`, the function reverts with `InsufficientOutput`.
 - ☐ Test coverage
- If `to.call` during the refund fails, the function reverts with `NativeTransferFailed`.

□ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).swap(recipient, zeroForOne, _toSignedExactInput(amountIn), priceLimitX64, "", "")` (via internal `_swapWithContext`)
 - **What is controllable?** `pool`, `recipient`, `zeroForOne`, `amountIn`, and `priceLimitX64` are caller-controlled. `callbackData` and `hookData` are fixed as empty values.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned deltas are decomposed by `_decodeSwapResult` and validated by the `minAmountOut` slippage guard.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped in `try/catch`; on revert, the swap context is cleared and the reason is bubbled up. During the swap, the pool calls back into `metricOmmSwapCallback` to wrap and settle the native input as WETH.
- `(bool ok, ) = to.call{value: msg.value - amountInUsed}("")` (`_refundUnusedNative`, refund of unused native)
 - **What is controllable?** `to` is the refund recipient (`msg.sender`) and the refund amount is computed as `msg.value - amountInUsed`.
 - **If the return value is controllable, how is it used and how can it go wrong?** If `ok == false`, the function reverts with `NativeTransferFailed`. The refund is performed before the swap context is cleared (`_clearSwap`).
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the refund recipient is a contract, `call` executes its code and reentrancy is possible; however, since the swap context is still active at this point, any reentrant swap on the same swapper is blocked by `SwapInProgress`.

Function: `swapExactOutputNativeForTokens(address pool, address recipient, bool zeroForOne, uint128 amountOutDesired, uint128 priceLimitX64, uint256 maxAmountIn, uint256 deadline)` external payable returns `(uint256 amountOut, uint256 amountInUsed)`

`swapExactOutputNativeForTokens` is the `exactOut` entry point that accepts native ETH as input to obtain an exact token output. Since the input ceiling is pre-paid as `msg.value`, it requires `msg.value == maxAmountIn` exactly. After swapping with `payerIsNative = true`, it checks output sufficiency (`amountOut >= amountOutDesired`, otherwise `InvalidSwapDeltas`) and input ceiling (`amountInUsed <= maxAmountIn`, otherwise `InputTooHigh`), then refunds unused native to the caller.

Inputs

- pool
 - **Control:** Arbitrary
 - **Constraints:** No whitelist.
 - **Impact:** The pool to swap against.
- recipient
 - **Control:** Arbitrary
 - **Constraints:** No address(0) validation.
 - **Impact:** Address to receive the output token.
- zeroForOne
 - **Control:** Arbitrary
 - **Constraints:** Direction flag. The input token must be WETH.
 - **Impact:** Determines the swap direction and input token.
- amountOutDesired
 - **Control:** Arbitrary
 - **Constraints:** If it exceeds `int128.max_in_toSignedExactOutput`, reverts with `AmountTooLarge`. Converted to a negative amount specification.
 - **Impact:** The desired exact output amount.
- priceLimitX64
 - **Control:** Arbitrary
 - **Constraints:** Direction-specific sentinel rules are the same.
 - **Impact:** Price limit for the swap.
- maxAmountIn
 - **Control:** Arbitrary
 - **Constraints:** `msg.value == maxAmountIn` is required (otherwise reverts with `InsufficientNativeValue`). Pre-paid ETH ceiling.
 - **Impact:** Serves as both the input ceiling and the refund reference amount. If `amountInUsed > maxAmountIn`, reverts with `InputTooHigh`.
- deadline
 - **Control:** Arbitrary
 - **Constraints:** If `block.timestamp > deadline`, reverts with `TransactionExpired`.
 - **Impact:** Upper time bound for the transaction.

Branches and code coverage

Intended branches

- When `msg.value == maxAmountIn`, the normal native `exactOut` swap executes.
 - ☒ Test coverage
- When `amountInUsed < maxAmountIn`, unused native is refunded to the caller.
 - ☒ Test coverage
- When `amountInUsed == maxAmountIn`, the refund amount is 0 and the function returns early.
 - ☐ Test coverage
- Internally, `_swapWithContext` is called with `payerIsNative=true` and `expectNativeOutput=false`. It performs price limit sentinel validation, transient swap context setup (`SwapInProgress`), `pool.swap` invocation, and `_decodeSwapResult` decomposition. During callback settlement, the swapper wraps its ETH into WETH and transfers it to the pool (see the `metricOmmSwapCallback` documentation). After the swap, if output sufficiency and input ceiling checks pass, unused native is refunded via `_refundUnusedNative`.
 - ☒ Test coverage
- `_swapWithContext` retains the swap context on success without clearing it; the wrapper calls `_clearSwap` after the refund. Therefore, the context remains active at the time of the refund to `.call`, so any reentrant swap attempt on the same swapper by the refund recipient reverts with `SwapInProgress`.
 - ☐ Test coverage

Negative branches

- If `block.timestamp > deadline`, the function reverts with `TransactionExpired`.
 - ☐ Test coverage
- If `msg.value != maxAmountIn`, the function reverts with `InsufficientNativeValue(maxAmountIn, msg.value)`.
 - ☐ Test coverage
- If the `priceLimitX64` sentinel does not match the direction, the function reverts with `InvalidPriceLimitForDirection`.
 - ☐ Test coverage
- If the input token is not WETH (within the callback), the function reverts with `NativeInputNotSupported`.
 - ☐ Test coverage
- If `amountOut < amountOutDesired`, the function reverts with `InvalidSwapDeltas`.
 - ☐ Test coverage
- If `amountInUsed > maxAmountIn`, the function reverts with `InputTooHigh(amountInUsed, maxAmountIn)`.
 - ☐ Test coverage

- If the refund to `.call` fails, the function reverts with `NativeTransferFailed`.

□ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).swap(recipient, zeroForOne, _toSignedExactOutput(amountOutDesired), priceLimitX64, "", "")` (via internal `_swapWithContext`)
 - **What is controllable?** `pool`, `recipient`, `zeroForOne`, `amountOutDesired`, and `priceLimitX64` are caller-controlled. `callbackData` and `hookData` are fixed as empty values.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned deltas are decomposed by `_decodeSwapResult` and validated for output sufficiency (`amountOut >= amountOutDesired`, otherwise `InvalidSwapDeltas`) and input ceiling (`amountInUsed <= maxAmountIn`, otherwise `InputTooHigh`).
 - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped in `try/catch`; on revert, the swap context is cleared and the reason is bubbled up. During the swap, the pool calls back into `metricOmmSwapCallback` to wrap and settle the native input as WETH.
- `(bool ok, ) = to.call{value: maxAmountIn - amountInUsed}("")` (`_refundUnusedNative`, refund of unused native)
 - **What is controllable?** `to` is the refund recipient (`msg.sender`) and the refund amount is `maxAmountIn - amountInUsed` (where `maxAmountIn == msg.value`).
 - **If the return value is controllable, how is it used and how can it go wrong?** `_refundUnusedNative` first reverts with `InputTooHigh` if `amountInUsed > maxAmountIn`, and returns early if the refund amount is 0. If `ok == false`, it reverts with `NativeTransferFailed`.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the refund recipient is a contract, reentrancy is possible; however, the swap context remains active, so any reentrant swap on the same swapper is blocked by `SwapInProgress`.

Function: swapExactInputTokensForNative(address pool, address recipient, bool zeroForOne, uint128 amountIn, uint128 priceLimitX64, uint256 minAmountOut, uint256 deadline) external returns (uint256 amountOut, uint256 amountInUsed)

swapExactInputTokensForNative is the exactIn entry point that accepts tokens as input and outputs native ETH (not payable). To receive the output WETH directly, the swapper sets recipient to address(this) and swaps with expectNativeOutput = true. The callback verifies that the output token is WETH (otherwise reverts with NativeOutputNotSupported), and once the slippage guard passes, the swapper unwraps the WETH and transfers native ETH to the actual recipient.

Inputs

- pool
 - **Control:** Arbitrary
 - **Constraints:** No whitelist.
 - **Impact:** The pool to swap against.
- recipient
 - **Control:** Arbitrary
 - **Constraints:** No address(0) validation. This is the final native ETH recipient; address(this) is passed to the pool as the recipient.
 - **Impact:** Address to receive the unwrapped ETH.
- zeroForOne
 - **Control:** Arbitrary
 - **Constraints:** Direction flag. The output token must be WETH.
 - **Impact:** Determines the swap direction and output token.
- amountIn
 - **Control:** Arbitrary
 - **Constraints:** If it exceeds int128.max in _toSignedExactInput, reverts with AmountTooLarge. The input token must be pre-approved by the payer (msg.sender) on the swapper for safeTransferFrom in the callback.
 - **Impact:** The exact token input amount.
- priceLimitX64
 - **Control:** Arbitrary
 - **Constraints:** Direction-specific sentinel rules are the same.
 - **Impact:** Price limit for the swap.
- minAmountOut
 - **Control:** Arbitrary

- **Constraints:** Slippage lower bound.
 - **Impact:** If `amountOut < minAmountOut`, the function reverts with `InsufficientOutput`.
- `deadline`
 - **Control:** Arbitrary
 - **Constraints:** If `block.timestamp > deadline`, reverts with `TransactionExpired`.
 - **Impact:** Upper time bound for the transaction.

Branches and code coverage

Intended branches

- The normal token-to-native exactIn swap succeeds and the WETH is unwrapped and sent.
 - ☒ Test coverage
- When `zeroForOne == true` or `false`, the output token is WETH in either case.
 - ☒ Test coverage
- Since this path is not payable, native input is not possible; the swapper unwraps the received WETH via `withdraw` and transfers native ETH to the recipient.
 - ☒ Test coverage
- Internally, `_swapWithContext` is called with `payerIsNative=false` and `expectNativeOutput=true`, with the pool recipient fixed to `address(this)`. It performs price limit sentinel validation, transient swap context setup (`SwapInProgress`), `pool.swap` invocation, `_decodeSwapResult` decomposition, and the `minAmountOut` slippage guard. The output WETH is held by the swapper and then unwrapped and sent to the actual recipient via `_unwrapAndSendNative`.
 - ☒ Test coverage
- `_swapWithContext` retains the context on success and the wrapper clears it with `_clearSwap` after unwrapping and sending. Therefore, the context is still active at the time of the `to.call` in `_unwrapAndSendNative`, so any reentrant swap attempt on the same swapper by the recipient reverts with `SwapInProgress`.
 - ☐ Test coverage

Negative branches

- If `block.timestamp > deadline`, the function reverts with `TransactionExpired`.
 - ☐ Test coverage
- If the `priceLimitX64` sentinel does not match the direction, the function reverts with `InvalidPriceLimitForDirection`.

- ☐ Test coverage
 - If the output token is not WETH (checked via `expectNativeOutput` in the callback), the function reverts with `NativeOutputNotSupported(outputToken)`.
- ☐ Test coverage
 - If `amountOut < minAmountOut`, the function reverts with `InsufficientOutput`.
- ☐ Test coverage
 - If `to.call` fails after unwrapping, the function reverts with `NativeTransferFailed`.
- ☐ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).swap(address(this), zeroForOne, _toSignedExactInput(amountIn), priceLimitX64, "", "")` (via internal `_swapWithContext`)
 - **What is controllable?** `pool`, `zeroForOne`, `amountIn`, and `priceLimitX64` are caller-controlled. The pool recipient is fixed to `address(this)`; `callbackData` and `hookData` are fixed as empty values.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned deltas are validated by `_decodeSwapResult` and the `minAmountOut` guard. The output WETH accumulates in the swapper, and the callback settles the input token from the payer.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped in `try/catch`; on revert, the context is cleared and the reason is bubbled up.
- `IWETH9(WETH).withdraw(amountOut) → (bool ok, ) = to.call{value: amountOut}("") (_unwrapAndSendNative)`
 - **What is controllable?** `amountOut` is the swap result and `to` is the final recipient specified by the caller.
 - **If the return value is controllable, how is it used and how can it go wrong?** `withdraw` returns ETH to the swapper's receive, which only accepts calls from WETH. The native ETH is then sent via `to.call`, and if `ok == false`, the function reverts with `NativeTransferFailed`.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If `to` is a contract, `call` executes its code and reentrancy is possible; however, the swap context remains active, so any reentrant swap on the same swapper is blocked by `SwapInProgress`.

Function: swapExactOutputTokensForNative(address pool, address recipient, bool zeroForOne, uint128 amountOutDesired, uint128 priceLimitX64, uint256 maxAmountIn, uint256 deadline) external returns (uint256 amountOut, uint256 amountInUsed)

swapExactOutputTokensForNative is the exactOut entry point that accepts tokens as input to obtain an exact native ETH output (not payable). With recipient set to address(this) and expectNativeOutput = true, it swaps, checks output sufficiency (amountOut >= amountOutDesired) and input ceiling (amountInUsed <= maxAmountIn), then unwraps the received WETH and transfers native ETH to the actual recipient.

Inputs

- pool
 - **Control:** Arbitrary
 - **Constraints:** No whitelist.
 - **Impact:** The pool to swap against.
- recipient
 - **Control:** Arbitrary
 - **Constraints:** No address(0) validation. address(this) is passed to the pool as the recipient.
 - **Impact:** Final address to receive the unwrapped ETH.
- zeroForOne
 - **Control:** Arbitrary
 - **Constraints:** Direction flag. The output token must be WETH.
 - **Impact:** Determines the swap direction and output token.
- amountOutDesired
 - **Control:** Arbitrary
 - **Constraints:** If it exceeds int128.max in _toSignedExactOutput, reverts with AmountTooLarge.
 - **Impact:** The desired exact native output amount.
- priceLimitX64
 - **Control:** Arbitrary
 - **Constraints:** Direction-specific sentinel rules are the same.
 - **Impact:** Price limit for the swap.
- maxAmountIn
 - **Control:** Arbitrary
 - **Constraints:** Slippage upper bound. If amountInUsed > maxAmountIn, reverts with InputTooHigh. The input token must be pre-approved.

- **Impact:** Maximum token expenditure ceiling.
- **deadline**
 - **Control:** Arbitrary
 - **Constraints:** If `block.timestamp > deadline`, reverts with `TransactionExpired`.
 - **Impact:** Upper time bound for the transaction.

Branches and code coverage

Intended branches

- The normal token-to-native exactOut swap succeeds and the WETH is unwrapped and sent.
 - ☒ Test coverage
- When `amountOut >= amountOutDesired` and `amountInUsed <= maxAmountIn`, the function returns normally.
 - ☒ Test coverage
- Internally, `_swapWithContext` is called with `payerIsNative=false` and `expectNativeOutput=true`, with the pool recipient fixed to `address(this)`. It performs price limit sentinel validation, transient swap context setup (`SwapInProgress`), `pool.swap` invocation, `_decodeSwapResult` decomposition, output sufficiency and input ceiling checks, and then unwraps the output WETH via `_unwrapAndSendNative` and transfers native ETH to the actual recipient.
 - ☒ Test coverage
- `_swapWithContext` retains the context on success and the wrapper clears it with `_clearSwap` after unwrapping and sending. Therefore, the context is still active at the time of the `to.call` in `_unwrapAndSendNative`, so any reentrant swap on the same swapper is blocked by `SwapInProgress`.
 - ☐ Test coverage

Negative branches

- If `block.timestamp > deadline`, the function reverts with `TransactionExpired`.
 - ☐ Test coverage
- If the `priceLimitX64` sentinel does not match the direction, the function reverts with `InvalidPriceLimitForDirection`.
 - ☐ Test coverage
- If the output token is not WETH (within the callback), the function reverts with `NativeOutputNotSupported(outputToken)`.
 - ☐ Test coverage

- If `amountOut < amountOutDesired`, the function reverts with `InvalidSwapDeltas`.
 - ☐ Test coverage
- If `amountInUsed > maxAmountIn`, the function reverts with `InputTooHigh(amountInUsed, maxAmountIn)`.
 - ☐ Test coverage
- If `to.call` fails after unwrapping, the function reverts with `NativeTransferFailed`.
 - ☐ Test coverage

Function call analysis

- `IMetricOmmPoolActions(pool).swap(address(this), zeroForOne, _toSignedExactOutput(amountOutDesired), priceLimitX64, "", "")` (via internal `_swapWithContext`)
 - **What is controllable?** `pool`, `zeroForOne`, `amountOutDesired`, and `priceLimitX64` are caller-controlled. The pool recipient is fixed to `address(this)`; `callbackData` and `hookData` are fixed as empty values.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned deltas are decomposed by `_decodeSwapResult` and validated for output sufficiency (`amountOut >= amountOutDesired`, otherwise `InvalidSwapDeltas`) and input ceiling (`amountInUsed <= maxAmountIn`, otherwise `InputTooHigh`).
 - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped in `try/catch`; on revert, the context is cleared and the reason is bubbled up. The callback settles the input token from the payer.
- `IWETH9(WETH).withdraw(amountOut) → (bool ok, ) = to.call{value: amountOut}("") (_unwrapAndSendNative)`
 - **What is controllable?** `amountOut` is the swap result and `to` is the final recipient specified by the caller.
 - **If the return value is controllable, how is it used and how can it go wrong?** `withdraw` returns ETH to the swapper's receive, which only accepts calls from WETH. The native ETH is then sent via `to.call`, and if `ok == false`, the function reverts with `NativeTransferFailed`.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If `to` is a contract, reentrancy is possible; however, the swap context remains active, so any reentrant swap on the same swapper is blocked by `SwapInProgress`.

Function: `metricOmmSwapCallback(int256 amount0Delta, int256 amount1Delta, bytes calldata) external override`

`metricOmmSwapCallback` is the callback invoked by the pool during swap execution to settle the input leg. The swapper loads (`payer`, `pool`, `flags`) from the transient swap context, verifies that `msg.sender == pool`, and reads `token0`/`token1` from the pool's immutables. If the `expectNativeOutput` flag is set, it verifies that the output token is WETH. For positive deltas (i.e., input tokens that the pool needs to receive), the swapper calls `_payInput` to transfer the token from the payer, or wraps the swapper's ETH into WETH and transfers it to the pool if native input is being used.

Inputs

- `amount0Delta`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Computed and passed by the pool. A positive value indicates that the pool needs to receive more token0.
 - **Impact:** Triggers token0 settlement when `> 0`.
- `amount1Delta`
 - **Control:** Arbitrary, but constrained by pool binding
 - **Constraints:** Computed and passed by the pool. A positive value indicates that token1 settlement is required.
 - **Impact:** Triggers token1 settlement when `> 0`.
- `bytes calldata (callbackData, unused)`
 - **Control:** Arbitrary
 - **Constraints:** Received without a name and not used in the function body.
 - **Impact:** None.

Branches and code coverage

Intended branches

- When `amount0Delta > 0`, token0 settlement via `_payInput` is executed.
 - ☒ Test coverage
- When `amount1Delta > 0`, token1 settlement is executed.
 - ☒ Test coverage
- When `payerIsNative == false`, the input token is transferred from the payer to the pool via `safeTransferFrom`.
 - ☒ Test coverage
- When `payerIsNative == true` and the input token is WETH, the swapper deposits its

ETH via deposit and transfers WETH to the pool via safeTransfer.

☒ Test coverage

- The payer is fixed as msg.sender at swap entry, so the callback cannot pull tokens from an arbitrary address — only funds approved by the payer itself are used.

☒ Test coverage

- Token settlement is performed via the internal _payInput. When payerIsNative == false, the single path is IERC20(token).safeTransferFrom(payer, pool, amount). When payerIsNative == true, it first verifies that the input token is WETH (NativeInputNotSupported), checks the swapper's balance (InsufficientNativeValue), then follows the WETH deposit and safeTransfer path.

☒ Test coverage

Negative branches

- If msg.sender != pool (does not match the context's pool), the function reverts with InvalidCallbackCaller.

☐ Test coverage

- If expectNativeOutput == true and the output token is not WETH, the function reverts with NativeOutputNotSupported(outputToken).

☐ Test coverage

- If payerIsNative == true and both amount0Delta > 0 and amount1Delta > 0 (both legs require input), the function reverts with InvalidSwapDeltas.

☐ Test coverage

- If payerIsNative == true but the input token is not WETH (in _payInput), the function reverts with NativeInputNotSupported(token).

☐ Test coverage

- If payerIsNative == true but the swapper's balance is insufficient (address(this).balance < amount), the function reverts with InsufficientNativeValue(amount, balance).

☐ Test coverage

- If the swap context is not set (direct call without an active swap), pool is address(0), so msg.sender != pool triggers InvalidCallbackCaller. Arbitrary calls outside an active swap are therefore blocked.

☐ Test coverage

Function call analysis

- IMetricOmmPool(pool).getImmutables()

- **What is controllable?** `pool` is the value stored in the swap context, specified by the caller at swap entry.
 - **If the return value is controllable, how is it used and how can it go wrong?** The `token0/token1` addresses from the returned `PoolImmutables` determine the tokens to settle. A non-standard pool could return incorrect token addresses, but the `msg.sender == pool` check ensures only the pool from the active swap can reach this point.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If this view call reverts, the callback fails and the entire `pool.swap` reverts.
- `IERC20(token).safeTransferFrom(payer, pool, amount) (_payInput, payerIsNative == false path)`
 - **What is controllable?** `payer` is the original caller fixed in the context; `amount` is the delta determined by the pool; `token` is from the pool's immutables.
 - **If the return value is controllable, how is it used and how can it go wrong?** Succeeds only within the limit pre-approved by the payer on the swapper. Fee-on-transfer tokens may result in the pool receiving less than expected, potentially causing delta or invariant validation failures on the pool side.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A transfer failure causes the callback to revert, rolling back the entire swap.
 - `IWETH9(WETH).deposit{value: amount}() → IERC20(WETH).safeTransfer(pool, amount) (_payInput, payerIsNative == true path)`
 - **What is controllable?** `amount` is the delta determined by the pool. The settlement token is fixed as `WETH`.
 - **If the return value is controllable, how is it used and how can it go wrong?** The swapper wraps its `ETH` (the `msg.value` forwarded at swap entry) into `WETH` and transfers it to the pool. If the swapper's balance is less than `amount`, `InsufficientNativeValue` blocks the operation in advance.
 - **What happens if it reverts, reenters, or does other unusual control flow?** A deposit or transfer failure causes the callback to revert, rolling back the entire swap.

Function: `getBidAndAskPrice()` external view override returns (uint128 bid, uint128 ask)

`getBidAndAskPrice` is the core view called by the pool to compute swap prices; it validates and processes off-chain oracle data and returns bid/ask in Q64 format. The internal `_getBidAndAskPrice` (1) reads `OracleData` from the oracle and converts `timestamps` to seconds, checks (2) staleness (including future timestamps) and (3) basic validity (`price != 0`, `spread < ORACLE_BPS`), (4) `priceGuard` bounds, (5) Chainlink deviation (`maxC1Deviation`), (6) confidence-weighted spread to derive bid/ask, (7) `cexStep` adjustment (`bid floor · ask ceil`), and (8) the `bid < ask` invariant. Any step failure causes the internal function to return the (0,

`type(uint128).max`) sentinel, and the external wrapper reverts with `FeedStalled` upon encountering this sentinel.

Branches and code coverage

Intended branches

- When all checks pass, the function returns `(uint128(bidOut), uint128(askOut))` successfully.
 - ☒ Test coverage
- When `maxClDeviation == CL_DEVIATION_CHECK_DISABLED(type(uint16).max)`, the Chainlink deviation check is skipped.
 - ☒ Test coverage
- If the Chainlink feed is unavailable (`!cloFound`) or stale, execution proceeds without the CL deviation check, preserving the previous behavior.
 - ☐ Test coverage
- When `cloFound && cloPrice_ != 0` and `data.price` falls within `cloPrice_ ± maxClDeviation`, the deviation check passes.
 - ☐ Test coverage
- When `guardMax == 0`, the upper bound is replaced with `type(uint128).max`, indicating no upper limit is configured.
 - ☒ Test coverage

Negative branches

- If `_isStale(refTime, now, MAX_TIME_DELTA)` returns true (or `refTime > now` is a future timestamp), the internal function returns the sentinel and the wrapper reverts with `FeedStalled`.
 - ☒ Test coverage
- If `data.price == 0` or `data.spread >= ORACLE_BPS`, the function returns the sentinel and reverts with `FeedStalled`.
 - ☒ Test coverage
- If `data.price < guardMin` or `data.price > guardMax`, the function returns the sentinel and reverts with `FeedStalled`.
 - ☒ Test coverage
- If the price falls outside the CL deviation bounds (`!inBound`), the function returns the sentinel and reverts with `FeedStalled`.
 - ☐ Test coverage
- If `_applyBidAdjustments` or `_applyAskAdjustments` returns `ok == false`, or the result exceeds `type(uint128).max`, the function returns the sentinel and reverts with

FeedStalled.

☐ Test coverage

- If `bidOut >= askOut` (which can occur when `cexStep` is negative combined with low confidence), the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- All failures are handled by returning the `(0, type(uint128).max)` sentinel rather than reverting directly; `FeedStalled` is only thrown by the external wrappers (`getBidAndAskPrice/getBidPrice/getAskPrice`). The UI variants `getBidAndAskPriceUi/getBidPriceUi/getAskPriceUi` expose the sentinel as-is.

☒ Test coverage

- If a Chainlink feed is decommissioned or frozen, a revert from the CL call in `_cloPriceInQuote` propagates and can brick the entire provider (noted in code comments). The mitigation is to call `setMaxClDeviation(CL_DEVIATION_CHECK_DISABLED)`.

☐ Test coverage

- The bid is rounded down (floor) and the ask is rounded up (ceil), ensuring the spread never narrows due to rounding direction.

☒ Test coverage

- The `_cloPriceInQuote` function (an internal of the inherited `ChainlinkVerifier`) used in the CL deviation check step reads two separate CL feeds for base and quote, applying `__sanityCheck(refPrice > 0 && updatedAt > 0 && answeredInRound >= roundId)` to each round. If either feed is not configured or fails the sanity check, `cloFound == false` is returned and the CL check is skipped. When valid, the prices are normalized to 8 decimals via `__normalizeToTargetDecimals`, and `refTime = min(base, quote updatedAt) * heartbeat = min(base, quote heartbeat)` are adopted. The returned `cloRefTime` is then re-checked for freshness via `_isStale(..., cloHeartbeat)`.

☐ Test coverage

Function call analysis

- `offchainOracle.getOracleData(offchainFeedId) / offchainOracle.priceGuard(offchainFeedId)` (via internal `_getBidAndAskPrice`)
 - **What is controllable?** `offchainOracle` and `offchainFeedId` are immutable. The returned price/spread/timestamp depend on the oracle (e.g., signature-based push).
 - **If the return value is controllable, how is it used and how can it go wrong?** The price/spread/timestamp go through staleness, validity, guard, and CL deviation checks before being processed into bid/ask values. If the oracle is manipulated, prices could be distorted, but the `priceGuard` and CL deviation checks serve as the primary defense.

- What happens if it reverts, reenters, or does other unusual control flow?**
 This is a view call; a revert causes the entire price query to fail.
- AggregatorV3Interface(baseOracle).latestRoundData() /
 AggregatorV3Interface(quoteOracle).latestRoundData() (via internal
 _cloPriceInQuote, for CL deviation reference computation)
- What is controllable?** The base/quote CL feed addresses are configured by the factory via setClOracle, and baseToken/quoteToken are immutable. The returned (roundId, answer, updatedAt, answeredInRound) depends on Chainlink.
- If the return value is controllable, how is it used and how can it go wrong?**
 The normalized CL reference price from both feeds sets the allowable deviation boundary (maxClDeviation) for the oracle price. If the CL feed is stale (exceeds heartbeat) or fails sanity, the check is skipped, so the CL heartbeat configuration determines the enforcement strength.
- What happens if it reverts, reenters, or does other unusual control flow?** If an individual CL feed call reverts (e.g., when decommissioned or frozen), the revert propagates and can brick the provider so that it can no longer return prices (as noted in code comments). The mitigation is to call setMaxClDeviation(CL_DEVIATION_CHECK_DISABLED).

Function: createPriceProvider(address _oracle, bytes32 _feedId, int256 _cexStep, uint256 _maxTimeDelta, uint16 _maxClDeviation, address _baseTokenOverride, address _quoteTokenOverride) external override returns (address provider)

createPriceProvider is a permissionless deployment function callable by anyone. It deploys a new PriceProvider with the given oracle/feed configuration and registers it in the factory. The new provider is created with the factory set to address(this), registered in the factory's _providers set and _providersByCreator[msg.sender], and providerOwner[provider] = msg.sender is set so that the caller becomes the owner. Parameter validation is mostly delegated to the PriceProvider constructor (e.g., _maxClDeviation range, _cexStep range, _maxTimeDelta zero/exceeds 7 days, base/quote zero address).

Inputs

- _oracle
 - Control:** Arbitrary
 - Constraints:** The factory does not validate this. The constructor calls IOffchainOracle(_oracle).getOracleInfo(_feedId), so a valid oracle is required for the base/quote tokens to be populated.
 - Impact:** The off-chain oracle from which the provider reads prices.

- `_feedId`
 - **Control:** Arbitrary
 - **Constraints:** No validation by the factory. The feed identifier used by the oracle.
 - **Impact:** Determines which feed's price is used.
- `_cexStep`
 - **Control:** Arbitrary
 - **Constraints:** The constructor reverts with `CexStepOutOfBounds` if `_cexStep <= -BPS_BASE || _cexStep >= BPS_BASE`.
 - **Impact:** The bid/ask step adjustment coefficient.
- `_maxTimeDelta`
 - **Control:** Arbitrary
 - **Constraints:** The constructor reverts with `MaxTimeDeltaOutOfBounds` if `0 > 7` days.
 - **Impact:** The allowed staleness window.
- `_maxClDeviation`
 - **Control:** Arbitrary
 - **Constraints:** The constructor reverts with `MaxClDeviationOutOfBounds` if `> ORACLE_BPS` and `!= CL_DEVIATION_CHECK_DISABLED`.
 - **Impact:** The Chainlink deviation tolerance (or the disabled sentinel value).
- `_baseTokenOverride / _quoteTokenOverride`
 - **Control:** Arbitrary
 - **Constraints:** If `address(0)`, the base/quote from oracle's `getOracleInfo` is used. If the final base/quote is zero, the constructor require fails.
 - **Impact:** Forces the provider's base/quote tokens to the specified addresses.

Branches and code coverage

Intended branches

- When all parameters are valid, the provider is deployed, registered in the set, owner is configured, and `ProviderDeployed` is emitted.
 - ☒ Test coverage
- When the override is zero, the base/quote from oracle info is adopted.
 - ☒ Test coverage

Negative branches

- If `_cexStep` falls outside the allowed range in the constructor, the function reverts with `CexStepOutOfBounds`.

- ☐ Test coverage
- If `_maxTimeDelta` falls outside the allowed range in the constructor, the function reverts with `MaxTimeDeltaOutOfBounds`.
- ☐ Test coverage
- If `_maxClDeviation` falls outside the allowed range in the constructor, the function reverts with `MaxClDeviationOutOfBounds`.
- ☐ Test coverage
- If the final base/quote is zero, the constructor require fails and the function reverts.
- ☐ Test coverage
- Since deployment is permissionless, anyone can create a provider pointing to an arbitrary oracle. However, registration in the factory tracking set alone does not grant trust to the provider in any pool; the pool factory must separately designate the `priceProvider` for it to be used.
- ☒ Test coverage
- The constructor validates inputs using the same range caps as setters such as `setConfidenceParam`, but `confidenceParam` starts at 0.
- ☐ Test coverage

Function call analysis

- `new PriceProvider(address(this), _oracle, _feedId, _cexStep, _maxTimeDelta, _maxClDeviation, _baseTokenOverride, _quoteTokenOverride)`
 - **What is controllable?** All arguments except the factory address are caller-controlled.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned address is registered in the tracking and owner mappings. The constructor internally calls `getOracleInfo`.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the constructor range validation or `getOracleInfo` fails, the entire deployment reverts.

Function: `function addProvider(address provider) external override on-lyRole(ADMIN_ROLE)`

`addProvider` is a function that allows an admin to manually register an already-deployed provider into the factory tracking set.

Inputs

- provider
 - **Control:** ADMIN_ROLE Only
 - **Constraints:** The provider must return `factory() == address(this)` and must not already be in the tracking set. This check does not prove deployed-by-this-factory provenance. No owner requirement is enforced.
 - **Impact:** The target of the permission and state change.

Branches and code coverage

Intended branches

- When an ADMIN_ROLE holder specifies a provider pointing to the current factory, the provider is added to the global tracking set.
 - ☒ Test coverage
- When `owner != address(0)`, the provider is added to both the global tracking set and the owner's per-creator provider index. If no owner is set, only the global tracking set is updated.
 - ☒ Test coverage

Negative branches

- If `PriceProvider(provider).factory() != address(this)`, the provider was created by a different factory and the function reverts.
 - ☐ Test coverage
- If `!_providers.add(provider)` is false, the provider is already being tracked and the function reverts with `ProviderAlreadyTracked()`.
 - ☐ Test coverage

Function call analysis

- `PriceProvider(provider).factory()`
 - **What is controllable?** The provider address is admin input.
 - **If the return value is controllable, how is it used and how can it go wrong?** If the returned factory address matches the current factory, the provider is added to tracking. The contract provenance of a spoofed return value is not separately verified.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the provider call reverts, the entire `addProvider` call reverts.

Function: createPriceProvider(address _oracle, bytes32 _feedId, int256 _cexStep, uint256 _maxTimeDelta, uint256 _futureTolerance, address _sequencerUptimeFeed, uint16 _maxC1Deviation, address _baseTokenOverride, address _quoteTokenOverride) external override returns (address provider)

createPriceProvider is the permissionless deployment function of the L2 factory. It accepts _futureTolerance (to allow sequencer clock skew) and _sequencerUptimeFeed (the sequencer uptime feed) in addition to the L1 parameters, and deploys a PriceProviderL2. The new provider is created with the factory set to address(this), registered in _providers and _providersByCreator[msg.sender], and providerOwner[provider] = msg.sender is set. Parameter validation (ranges, etc.) is delegated to the PriceProviderL2 constructor. On L2, an additional check rejects _futureTolerance > 1 hours with FutureToleranceOutOfBounds.

Inputs

- **_oracle**
 - **Control:** Arbitrary
 - **Constraints:** Not validated by the factory. The constructor calls getOracleInfo(_feedId).
 - **Impact:** The off-chain oracle from which prices are read.
- **_feedId**
 - **Control:** Arbitrary
 - **Constraints:** Not validated by the factory.
 - **Impact:** The feed to be used.
- **_cexStep**
 - **Control:** Arbitrary
 - **Constraints:** The constructor reverts with CexStepOutOfBounds on range violation.
 - **Impact:** The step adjustment coefficient.
- **_maxTimeDelta**
 - **Control:** Arbitrary
 - **Constraints:** The constructor reverts with MaxTimeDeltaOutOfBounds if 0 or > 7 days.
 - **Impact:** The allowed staleness window.
- **_futureTolerance**
 - **Control:** Arbitrary

- **Constraints:** The constructor reverts with `FutureToleranceOutOfBounds` if `> 1` hours.
 - **Impact:** The allowed future timestamp window to handle sequencer clock skew.
- `_sequencerUptimeFeed`
 - **Control:** Arbitrary
 - **Constraints:** Stored as-is in the `ChainlinkVerifierL2` constructor. If `address(0)`, `_checkSequencer` always returns healthy (true).
 - **Impact:** The feed used to determine sequencer downtime and grace period status.
- `_maxClDeviation`
 - **Control:** Arbitrary
 - **Constraints:** Range validation is performed by the constructor.
 - **Impact:** The Chainlink deviation tolerance (or the disabled sentinel value).
- `_baseTokenOverride` / `_quoteTokenOverride`
 - **Control:** Arbitrary
 - **Constraints:** If zero, oracle info base/quote is used. If the final value is zero, the require fails.
 - **Impact:** Forces the base/quote token addresses.

Branches and code coverage

Intended branches

- When all parameters are valid, `PriceProviderL2` is deployed, registered, owner is configured, and `ProviderDeployed` is emitted.
 - ☒ Test coverage
- When the override is zero, the base/quote from oracle info is adopted.
 - ☒ Test coverage
- When `_sequencerUptimeFeed == address(0)` is used for deployment, the sequencer check is disabled.
 - ☒ Test coverage

Negative branches

- If `_cexStep` falls outside the allowed range, the function reverts with `CexStepOutOfBounds`.
 - ☐ Test coverage
- If `_maxTimeDelta` falls outside the allowed range, the function reverts with `MaxTimeDeltaOutOfBounds`.

- ☐ Test coverage
 - If `_futureTolerance > 1` hours, the function reverts with `FutureToleranceOutOfBounds`.
- ☐ Test coverage
 - If `_maxClDeviation` falls outside the allowed range, the function reverts with `MaxClDeviationOutOfBounds`.
- ☐ Test coverage
 - If the final base/quote is zero, the require fails and the function reverts.
- ☐ Test coverage
 - Deploying with `_sequencerUptimeFeed` set to `address(0)` causes `_checkSequencer` to always return true, removing sequencer downtime protection even on L2. Any deployer can make this choice freely (permissionless).
- ☐ Test coverage

Function call analysis

- `new PriceProviderL2(address(this), _oracle, _feedId, _cexStep, _maxTimeDelta, _futureTolerance, _sequencerUptimeFeed, _maxClDeviation, _baseTokenOverride, _quoteTokenOverride)`
 - **What is controllable?** All arguments except the factory address are caller-controlled.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned address is registered in the tracking and owner mappings. The constructor calls `getOracleInfo` and validates range and futureTolerance.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the constructor validation or `getOracleInfo` fails, the deployment reverts.

Function: `addProvider(address provider) external override only-Role(ADMIN_ROLE) returns ()`

`addProvider` is a function that allows `ADMIN_ROLE` to register an externally created `PriceProviderL2` into the tracking set. It requires `PriceProviderL2(provider).factory() == address(this)`, preventing providers not deployed by this L2 factory from being added. If the provider is not newly added, the function reverts with `ProviderAlreadyTracked`. If `providerOwner` exists, the creator index is also updated (same structure as the L1 factory, but bound to `PriceProviderL2`).

Inputs

- provider
 - **Control:** ADMIN_ROLE Only (target address is arbitrary)
 - **Constraints:** PriceProviderL2(provider).factory() == address(this) is required. If already tracked, reverts with ProviderAlreadyTracked.
 - **Impact:** Exposed to the tracking set, batch setters, and view functions.

Branches and code coverage

Intended branches

- When the factory binding matches and the provider is not yet tracked, the provider is added to the set, the creator index is updated if an owner exists, and ProviderAdded is emitted.
 - ☒ Test coverage
- When providerOwner[provider] == address(0), the creator index update is skipped.
 - ☐ Test coverage

Negative branches

- If the caller does not hold ADMIN_ROLE, the function reverts via AccessControl.
 - ☐ Test coverage
- If PriceProviderL2(provider).factory() != address(this), the require fails and the function reverts.
 - ☐ Test coverage
- If the provider is already being tracked, the function reverts with ProviderAlreadyTracked.
 - ☐ Test coverage
- The factory binding check prevents insertion of L2 providers deployed by other factories. A provider without an owner can also be added, but in that case there may be no entity authorized to pass the updater-permission-based batch setter.
 - ☐ Test coverage

Function call analysis

- PriceProviderL2(provider).factory()
 - **What is controllable?** The provider address is admin input.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned factory must equal address(this). While the return value can

be spoofed, this operation occurs within the trust boundary of a direct ADMIN action.

- **What happens if it reverts, reenters, or does other unusual control flow?** If the view call reverts, the addition fails.

Function: `getBidAndAskPrice()` external view override returns `(uint128 bid, uint128 ask)`

`getBidAndAskPrice` is a view that provides bid/ask prices to the pool via the L2 price provider, following the same 8-step pipeline as the L1 variant, with two L2-specific additions. First, the staleness check accepts a `FUTURE_TOLERANCE` parameter, allowing the oracle `refTime` to be up to `FUTURE_TOLERANCE` seconds in the future relative to `block.timestamp` (to handle sequencer clock skew). Second, the Chainlink deviation check (step 5) goes through `ChainlinkVerifierL2._cloPriceInQuote`, which first calls `_checkSequencer()` and returns `(false, ...)` during sequencer downtime or the recovery grace period, thereby **gracefully skipping the CL deviation check**. All internal failures return the `(0, type(uint128).max)` sentinel, and the external wrapper reverts with `FeedStalled` upon encountering it.

Branches and code coverage

Intended branches

- When all checks pass, the function returns `(uint128(bidOut), uint128(askOut))` successfully.
 - ☒ Test coverage
- When `refTime > now` but `refTime - now <= FUTURE_TOLERANCE`, the future timestamp is permitted and the check passes.
 - ☒ Test coverage
- When `maxClDeviation == CL_DEVIATION_CHECK_DISABLED`, the Chainlink deviation check is skipped.
 - ☒ Test coverage
- When the sequencer is healthy (`_checkSequencer() == true`) and the CL feed is valid and in-bound, the deviation check passes.
 - ☐ Test coverage
- If the sequencer is down or the recovery grace period has not elapsed, `_cloPriceInQuote` returns `(false, ...)` and execution proceeds without the CL check.
 - ☐ Test coverage
- If the CL feed is not configured or is stale, execution proceeds without the CL check.

☐ Test coverage

Negative branches

- If `_isStale(refTime, now, MAX_TIME_DELTA, FUTURE_TOLERANCE)` returns true (excessively old in the past, or more than `FUTURE_TOLERANCE` in the future), the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- If `data.price == 0` or `data.spread >= ORACLE_BPS`, the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- If `data.price` falls outside the `priceGuard` bounds, the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- If the price falls outside the CL deviation bounds (`!inBound`), the function returns the sentinel and reverts with `FeedStalled`.

☐ Test coverage

- If the bid/ask step adjustment fails or the result exceeds `uint128`, the function returns the sentinel and reverts with `FeedStalled`.

☐ Test coverage

- If `bidOut >= askOut`, the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- The CL deviation check is **disabled during the sequencer grace period**: `_checkSequencer` returns false if the sequencer has just recovered and `block.timestamp - startedAt <= GRACE_PERIOD`. In that case, `_cloPriceInQuote` immediately returns `(false, 0, 0, 0)`, causing the step-5 CL guard to be bypassed entirely. This means that during the grace period, off-chain oracle prices can pass through without Chainlink deviation verification (related to finding #100).

☐ Test coverage

- During sequencer downtime, `_checkSequencer` returns false when `answer != 0`, similarly skipping the CL check. The omission of the CL check is handled as 'proceed without check' rather than a revert.

☐ Test coverage

- `_isStale` compares past and future timestamps using only subtraction to avoid overflow. Unlike the L1 variant, it accepts the `FUTURE_TOLERANCE` argument.

☒ Test coverage

- The `_cloPriceInQuote` function (an internal of the inherited `ChainlinkVerifierL2`) used in the CL step first calls `_checkSequencer()`, then reads both the base and quote CL feeds and applies `__sanityCheck(answer > 0 && updatedAt > 0; unlike the L1 variant, answeredInRound is not compared)` to each round. If either feed is unconfigured

or fails the sanity check, `cloFound == false` is returned and the CL check is skipped. When valid, prices are normalized to 8 decimals, and `refTime = min(base, quote) · heartbeat = min(base, quote)` are adopted. The returned `cloRefTime` is re-checked for freshness via `_isStale(..., cloHeartbeat)`.

□ Test coverage

Function call analysis

- `offchainOracle.getOracleData(offchainFeedId) / offchainOracle.priceGuard(offchainFeedId)` (via internal `_getBidAndAskPrice`)
 - **What is controllable?** `offchainOracle` and `offchainFeedId` are immutable. The returned price/spread/timestamp depend on the oracle.
 - **If the return value is controllable, how is it used and how can it go wrong?** The values go through staleness, validity, guard, and CL deviation checks as in the L1 variant. However, since L2 allows future timestamps up to `FUTURE_TOLERANCE`, a slightly future timestamp exploiting sequencer skew can pass through.
 - **What happens if it reverts, reenters, or does other unusual control flow?** This is a view call; a revert causes the entire price query to fail.
- `sequencerUptimeFeed.latestRoundData()` (via internal `_cloPriceInQuote → _checkSequencer`, for sequencer status check)
 - **What is controllable?** The sequencer feed address is configured by the deployer/factory (address(0) is treated as always healthy). The returned (answer, startedAt) depends on the L2 sequencer uptime feed.
 - **If the return value is controllable, how is it used and how can it go wrong?** When `answer != 0` (down), `startedAt > now`, or the grace period has not elapsed (`now - startedAt <= GRACE_PERIOD`), the function returns `false` and **the entire CL deviation guard is gracefully bypassed** (related to finding #100). This means off-chain prices can pass through without Chainlink deviation verification during the grace period.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If this view reverts, the CL check immediately causes the entire price query to revert.
- `AggregatorV3Interface(baseOracle).latestRoundData() / AggregatorV3Interface(quoteOracle).latestRoundData()` (via internal `_cloPriceInQuote`, for CL deviation reference computation)
 - **What is controllable?** The base/quote CL feed addresses are configured by the factory via `setC1Oracle`, and `baseToken/quoteToken` are immutable. Return values depend on Chainlink.
 - **If the return value is controllable, how is it used and how can it go wrong?** The normalized CL reference price from both feeds sets the allowable deviation boundary (`maxC1Deviation`) for the oracle price. If the CL feed is

stale (exceeds heartbeat) or fails sanity, the check is skipped.

- **What happens if it reverts, reenters, or does other unusual control flow?** If an individual CL feed reverts due to being decommissioned or frozen, the revert propagates and can brick the provider (as noted in code comments). Sequencer downtime is handled gracefully, but individual feed reverts are a separate concern.

Function: `getBidAndAskPrice()` external returns (uint128 bid, uint128 ask)

`getBidAndAskPrice` is the swap-time read entry point for the abuse-protected oracle (Pyth/Chainlink providers) price provider. Unlike the standard `PriceProvider`, it is **non-view** and reads data exclusively through the oracle's attributed `price(feedId, pool)` path, with no access to public `getOracleData`. When the pool marks itself as in-swap with this provider and calls this function without arguments, the provider forwards `msg.sender` (the pool) to the oracle as `price(offchainFeedId, msg.sender)`, and the oracle validates and records the read (`PriceRead` event) via the `pool.inSwap() == provider binding`. The returned (`mid`, `spread`, `refTime`) goes through the staleness, price guard, CL deviation, confidence spread, and `cexStep` pipeline in `_computeBidAsk`, and on failure the (`0`, `type(uint128).max`) sentinel causes a `FeedStalled` revert.

Branches and code coverage

Intended branches

- When all checks on the oracle.price return value pass, the function returns a valid (bid, ask).
☒ Test coverage
- When `maxClDeviation == CL_DEVIATION_CHECK_DISABLED`, the Chainlink deviation check is skipped.
☒ Test coverage
- If the CL feed is not configured or is stale, execution proceeds without the CL check.
☐ Test coverage
- When `guardMax == 0`, the upper bound is replaced with `type(uint128).max`.
☒ Test coverage

Negative branches

- If the `refTime` from oracle.price is stale or in the future (on L1, all future timestamps are treated as stale), the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- If `price == 0` or `spread >= ORACLE_BPS`, the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- If `price` falls outside the `priceGuard` bounds, the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- If the price falls outside the CL deviation bounds, the function returns the sentinel and reverts with `FeedStalled`.

☐ Test coverage

- If the bid/ask step adjustment fails or the result exceeds `uint128`, the function returns the sentinel and reverts with `FeedStalled`.

☐ Test coverage

- If `bidOut >= askOut`, the function returns the sentinel and reverts with `FeedStalled`.

☒ Test coverage

- Since this function is non-view, state changes occur along the `oracle.price` path (such as recording `PriceRead` events). As a result, static integrations that call this function as a view are not possible; it only behaves as intended in an actual in-swap context.

☒ Test coverage

- The key distinction from a standard `PriceProvider` is that data is sourced from the attributed `price(feedId, pool)` rather than `getOracleData`. This attributes reads to a specific pool, enabling abuse tracking and blocking.

☒ Test coverage

- Post-read processing is handled by `_computeBidAsk(mid, spread, refTime)`, which applies the same staleness, price guard, CL deviation, confidence, and `cexStep` pipeline as the standard `PriceProvider`. The `_cloPriceInQuote` function (an internal of the inherited `ChainlinkVerifier`) in the CL step reads two CL feeds (base and quote), applies `__sanityCheck` to each round, and skips the CL check with `cloFound == false` if either feed is unconfigured or fails sanity. When valid, prices are normalized to 8 decimals, and `refTime = min(base, quote) * heartbeat = min(base, quote)` are adopted. The returned `cloRefTime` is re-checked for freshness via `_isStale(..., cloHeartbeat)`.

☐ Test coverage

Function call analysis

- `IPricedOracle(address(offchainOracle)).price(offchainFeedId, msg.sender)` (via internal `_getBidAndAskPrice`)

- **What is controllable?** `offchainFeedId` is immutable, and `msg.sender` is the calling pool. The provider does not exercise additional control over the arguments.
- **If the return value is controllable, how is it used and how can it go wrong?** The returned `(mid, spread, _, refTime)` serves as the input for price computation. If the oracle returns incorrect values, the `priceGuard` and CL deviation checks serve as the primary defense.
- **What happens if it reverts, reenters, or does other unusual control flow?** This is a non-view call; it reverts if the oracle-side binding (`pool.inSwap() == provider`), `blacklist`, or `feedExists` checks are not passed. If the oracle attempts reentrancy, the pool-side `nonReentrant` guard activates.
- `offchainOracle.priceGuard(offchainFeedId)` (via internal `_computeBidAsk`)
 - **What is controllable?** `offchainOracle` and `offchainFeedId` are immutable. The returned `(guardMin, guardMax)` depends on oracle configuration.
 - **If the return value is controllable, how is it used and how can it go wrong?** When `guardMax == 0`, the upper bound is replaced with `type(uint128).max`. If the oracle price falls outside `[guardMin, guardMax]`, it is handled as a sentinel. A loose guard may allow abnormal prices to pass through.
 - **What happens if it reverts, reenters, or does other unusual control flow?** This is a view call; a revert causes the entire price query to fail.
- `AggregatorV3Interface(baseOracle).latestRoundData() / AggregatorV3Interface(quoteOracle).latestRoundData()` (via internal `_cloPriceInQuote`, for CL deviation reference computation)
 - **What is controllable?** The base/quote CL feed addresses are configured by the factory via `setClOracle`, and `baseToken/quoteToken` are immutable. Return values depend on Chainlink.
 - **If the return value is controllable, how is it used and how can it go wrong?** The normalized CL reference price from both feeds sets the allowable deviation boundary for the oracle price. If the CL feed is stale or fails sanity, the check is skipped.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If an individual CL feed reverts, the revert propagates and can brick the provider (as noted in code comments). The mitigation is to call `setMaxClDeviation(CL_DEVIATION_CHECK_DISABLED)`.

Function: `getBidAndAskPrice()` external returns `(uint128 bid, uint128 ask)`

`getBidAndAskPrice` is the swap-time read entry point for the L2 abuse-protected price provider. Like `ProtectedPriceProvider`, it is **non-view** and reads data exclusively through the oracle's

attributed `price(feedId, msg.sender)` path (no public `getOracleData`). As L2-specific additions: (1) the staleness check accepts `FUTURE_TOLERANCE`, allowing the oracle `refTime` to be up to `FUTURE_TOLERANCE` seconds in the future; and (2) the CL deviation check gracefully bypasses via `ChainlinkVerifierL2._cloPriceInQuote` calling `_checkSequencer()` during sequencer downtime or the recovery grace period. All failures are handled with the `(0, type(uint128).max)` sentinel and the wrapper reverts with `FeedStalled`.

Branches and code coverage

Intended branches

- When all checks pass, the function returns a valid `(bid, ask)`.
 - ☐ Test coverage
- When `refTime > now` but `refTime - now <= FUTURE_TOLERANCE`, the future timestamp is permitted.
 - ☐ Test coverage
- When `maxClDeviation == CL_DEVIATION_CHECK_DISABLED`, the Chainlink deviation check is skipped.
 - ☐ Test coverage
- If the sequencer is down or the recovery grace period has not elapsed, `_cloPriceInQuote` returns `(false, ...)` and the CL check is skipped.
 - ☐ Test coverage
- If the CL feed is not configured or is stale, the CL check is skipped.
 - ☐ Test coverage

Negative branches

- If `_isStale(refTime, now, MAX_TIME_DELTA, FUTURE_TOLERANCE)` returns true, the function returns the sentinel and reverts with `FeedStalled`.
 - ☐ Test coverage
- If `price == 0` or `spread >= ORACLE_BPS`, the function returns the sentinel and reverts with `FeedStalled`.
 - ☐ Test coverage
- If price falls outside the priceGuard bounds, the function returns the sentinel and reverts with `FeedStalled`.
 - ☐ Test coverage
- If the price falls outside the CL deviation bounds, the function returns the sentinel and reverts with `FeedStalled`.
 - ☐ Test coverage
- If the bid/ask step adjustment fails or the result exceeds `uint128`, the function returns

the sentinel and reverts with FeedStalled.

☐ Test coverage

- If `bidOut >= askOut`, the function returns the sentinel and reverts with FeedStalled.

☐ Test coverage

- During the sequencer grace period, the entire CL deviation guard is bypassed, allowing off-chain oracle prices to pass through without Chainlink deviation verification (related to finding #100). Additionally, L2 permits future timestamps up to FUTURE_TOLERANCE.

☐ Test coverage

- Since this function is non-view, state changes occur along the oracle.price path (such as recording PriceRead events), and the attributed read binding enables pool-specific tracking.

☐ Test coverage

- Post-read processing is handled by `_computeBidAsk(mid, spread, refTime)` (the same pipeline as ProtectedPriceProvider plus L2 staleness/sequencer handling). The `_cloPriceInQuote` function (an internal of the inherited ChainlinkVerifierL2) in the CL step first calls `_checkSequencer()`, then reads both CL feeds (base and quote) and applies `__sanityCheck(answer > 0 && updatedAt > 0)`. If either feed is unconfigured or fails sanity, `cloFound == false` is returned and the CL check is skipped. When valid, prices are normalized to 8 decimals, and `refTime = min(heartbeat, min)` are adopted, with `cloRefTime` re-checked via `_isStale(..., cloHeartbeat)`.

☐ Test coverage

Function call analysis

- `IPricedOracle(address(offchainOracle)).price(offchainFeedId, msg.sender)` (via internal `_getBidAndAskPrice`)
 - **What is controllable?** `offchainFeedId` is immutable, and `msg.sender` is the calling pool.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned `(mid, spread, _, refTime)` serves as the input for price computation. The priceGuard and CL deviation checks serve as the primary defense.
 - **What happens if it reverts, reenters, or does other unusual control flow?** This is a non-view call; it reverts if the oracle-side in-swap binding (`pool.inSwap() == provider`), blacklist, or feedExists checks are not passed.
- `offchainOracle.priceGuard(offchainFeedId)` (via internal `_computeBidAsk`)
 - **What is controllable?** `offchainOracle` and `offchainFeedId` are immutable. The returned `(guardMin, guardMax)` depends on oracle configuration.

- **If the return value is controllable, how is it used and how can it go wrong?**
When `guardMax == 0`, the upper bound is replaced with `type(uint128).max`. If the price falls outside `[guardMin, guardMax]`, it is handled as a sentinel.
- **What happens if it reverts, reenters, or does other unusual control flow?**
This is a view call; a revert causes the entire price query to fail.
- `sequencerUptimeFeed.latestRoundData()` (via internal `_cloPriceInQuote` → `_checkSequencer`, for sequencer status check)
 - **What is controllable?** The sequencer feed address is configured by the deployer/factory (`address(0)` is treated as always healthy). The returned (`answer`, `startedAt`) depends on the sequencer uptime feed.
 - **If the return value is controllable, how is it used and how can it go wrong?**
When `answer != 0` (down) or the grace period has not elapsed, the function returns `false` and the entire CL deviation guard is gracefully bypassed (related to finding #100).
 - **What happens if it reverts, reenters, or does other unusual control flow?** If this view reverts, the entire price query reverts.
- `AggregatorV3Interface(baseOracle).latestRoundData()` / `AggregatorV3Interface(quoteOracle).latestRoundData()` (via internal `_cloPriceInQuote`, for CL deviation reference computation)
 - **What is controllable?** The base/quote CL feed addresses are configured by the factory via `setC1Oracle`, and `baseToken/quoteToken` are immutable.
 - **If the return value is controllable, how is it used and how can it go wrong?**
The normalized CL reference price from both feeds sets the allowable deviation boundary for the oracle price. If the CL feed is stale or fails sanity, the check is skipped.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If an individual CL feed reverts, the revert propagates and can brick the provider (sequencer downtime is handled gracefully as a separate matter).

Function: `function registerFeeds( bytes32[] calldata pythLazerFeedIds, address[] calldata baseTokens, address[] calldata quoteTokens ) external onlyRole(REGISTRAR_ROLE)`

`registerFeeds` is the function by which the registrar batch-registers multiple Pyth Lazer feed IDs and their corresponding token pairs.

Inputs

- `pythLazerFeedIds`

- **Control:** REGISTRAR_ROLE Only
 - **Constraints:** Must have the same length as the token arrays, and each `uint256(pythLazerFeedId)` must be within the range `< type(uint32).max`.
 - **Impact:** Serves as the key for Pyth feed metadata.
- `baseTokens`
 - **Control:** REGISTRAR_ROLE Only
 - **Constraints:** Must have the same length as the feed ID array; each base token must be nonzero and differ from the quote token at the same index.
 - **Impact:** Used as provider token0 / the base asset.
- `quoteTokens`
 - **Control:** REGISTRAR_ROLE Only
 - **Constraints:** Must have the same length as the feed ID array; each quote token must be nonzero and differ from the base token at the same index.
 - **Impact:** Used as provider token1 / the quote asset.

Branches and code coverage

Intended branches

- If the caller holds REGISTRAR_ROLE and all three arrays have the same length, the function passes `pythLazerFeedIds[i]`, `baseTokens[i]`, and `quoteTokens[i]` to `_registerFeed` for each index `i`, adds the feed ID to the registry, and stores the base/quote token metadata.
 - ☒ Test coverage
- When `_addFeedId` succeeds for each entry, `oracleData[...].timestampMs` is initialized to the sentinel `toTimeMs(1)`, so the first Lazer update after registration must satisfy `isAfter(1ms)` (strict `>`) to be stored. At registration time, `min/max price (0)` and `stateGuard(address(0))` are not set.
 - ☐ Test coverage
- If all three arrays have length 0, the loop does not execute and the function succeeds without adding any feeds.
 - ☒ Test coverage
- Entries within the batch are processed sequentially starting from the first index, but because the transaction is atomic, the final state is updated only if all entries pass.
 - ☐ Test coverage

Negative branches

- If the caller does not have REGISTRAR_ROLE, the `OpenZeppelin onlyRole(REGISTRAR_ROLE)` check reverts before the length check and loop execute.
 - ☐ Test coverage

- If any of `pythLazerFeedIds.length`, `baseTokens.length`, or `quoteTokens.length` differ, the function reverts with `LengthMismatch()`, leaving no feeds registered.
 - ☐ Test coverage
- If any entry in the batch has `uint256(pythLazerFeedId) >= type(uint32).max`, the function reverts with `LazerIdOverflow(pythLazerFeedId)`, rolling back the entire batch including that entry.
 - ☐ Test coverage
- If `_addFeedId(pythLazerFeedId)` returns false for any entry in the batch, the function reverts with `FeedAlreadyExists(pythLazerFeedId)`, rolling back the entire batch that includes the duplicate feed.
 - ☐ Test coverage
- If any entry in the batch has `baseToken == address(0)`, `quoteToken == address(0)`, or `baseToken == quoteToken`, the function reverts during token metadata assignment, rolling back the entire batch.
 - ☐ Test coverage
- Base/quote token metadata is trusted registrar/admin input only. The code verifies nonzero and distinct values but does not prove that the Pyth feed semantically corresponds to the given pair or direction.
 - ☐ Test coverage
- Registration emits a token-metadata event and a feed-registration event with different semantics. `FeedRegistered` carries placeholder guard/price-guard fields, not actual token updates.
 - ☐ Test coverage

Function: `function registerFeed( bytes32 pythLazerFeedId, address baseToken, address quoteToken ) public onlyRole(REGISTRAR_ROLE)`

`registerFeed` is the function by which the registrar registers a single Pyth Lazer feed ID and its corresponding token pair.

Inputs

- `pythLazerFeedId`
 - **Control:** `REGISTRAR_ROLE` Only
 - **Constraints:** Must be within the range `uint256(pythLazerFeedId) < type(uint32).max`; reverts if a duplicate registration is attempted.
 - **Impact:** The oracle feed key being registered.

- `baseToken`
 - **Control:** REGISTRAR_ROLE Only
 - **Constraints:** Must not be the zero address and must differ from the quote token; stored as internal token metadata.
 - **Impact:** Used as the token0 / base asset of the pool/provider.
- `quoteToken`
 - **Control:** REGISTRAR_ROLE Only
 - **Constraints:** Must not be the zero address and must differ from the base token; stored as internal token metadata.
 - **Impact:** Used as the token1 / quote asset of the pool/provider.

Branches and code coverage

Intended branches

- If the caller holds REGISTRAR_ROLE, `uint256(pythLazerFeedId) < type(uint32).max`, and the feed is not yet registered, the feed ID is added to the registry and `baseToken/quoteToken` metadata is stored.

☒ Test coverage

- When `_addFeedId` succeeds, `oracleData[pythLazerFeedId].timestampMs` is initialized to the sentinel `toTimeMs(1)`, so the first Lazer update after registration must satisfy `isAfter(1ms)` (strict >) to be stored. At registration time, `min/max price(0)` and `stateGuard(address(0))` are not set.

☐ Test coverage

Negative branches

- If the caller does not have REGISTRAR_ROLE, the `OpenZeppelin onlyRole(REGISTRAR_ROLE)` check reverts before `_registerFeed` executes.

☐ Test coverage

- If `uint256(pythLazerFeedId) >= type(uint32).max`, the function reverts with `LazerIdOverflow(pythLazerFeedId)`, leaving the registry and token metadata unchanged.

☐ Test coverage

- If `_addFeedId(pythLazerFeedId)` returns false, the function reverts with `FeedAlreadyExists(pythLazerFeedId)`, preventing duplicate registration of the same Pyth Lazer feed ID.

☒ Test coverage

- If `baseToken == address(0)`, `quoteToken == address(0)`, or `baseToken == quoteToken`, the function reverts during token metadata assignment.

- ☐ Test coverage
 - Base/quote token metadata is trusted registrar/admin input only. The code verifies nonzero and distinct values but does not prove that the Pyth feed semantically corresponds to the given pair or direction.
- ☐ Test coverage
 - Registration emits a token-metadata event and a feed-registration event with different semantics. FeedRegistered carries placeholder guard/price-guard fields, not actual token updates.
- ☐ Test coverage

Function: fallback() payable external override

fallback is a payable update entrypoint that receives a Pyth Lazer price update payload and passes it to `_verifyAndStore` for verification and storage. The calldata format is `[deadline:4][feedsLength:2][feedIds:feedsLength*4][priceUpdate:rest]`.

Inputs

- `deadline(calldata[0:4])`
 - **Control:** Arbitrary
 - **Constraints:** `block.timestamp <= deadline` must hold; equality is permitted.
 - **Impact:** Prevents the use of expired price update payloads.
- `feedsLength(calldata[4:6])`
 - **Control:** Arbitrary
 - **Constraints:** Decoded as a 2-byte length; determines the length of the subsequent feed ID slice.
 - **Impact:** Determines the number of feed IDs to decode.
- `feedIds(calldata[6:6+feedsLength*4])`
 - **Control:** Arbitrary
 - **Constraints:** Each feed ID is decoded as a 4-byte Pyth Lazer feed ID and passed to `_verifyAndStore`.
 - **Impact:** Determines which feeds' oracle data will be verified and stored from the update payload.
- `priceUpdate(calldata[6+feedsLength*4:calldatasize])`
 - **Control:** Arbitrary
 - **Constraints:** Must pass `_verifyAndStore` verification from LazerConsumer.

- **Impact:** Verified price, spread, and timestamp data are reflected in oracle storage.
- `msg.value`
 - **Control:** Arbitrary
 - **Constraints:** The fallback is payable with no separate refund logic. The Pyth verification fee is paid as a fixed cost from the contract balance, and ETH attached to the same call is included in that balance.
 - **Impact:** ETH sent by the caller may fund the verification fee payment; any balance remaining after the fee stays in the contract.

Branches and code coverage

Intended branches

- If the deadline is valid, the function decodes `feedsLength` 4-byte feed IDs, slices the remaining `calldata` as `priceUpdate`, and passes them to `_verifyAndStore`.

☒ Test coverage

Negative branches

- If `feedsLength == 0`, the `calldata` feed list is empty. If the number of feeds in the verified payload differs, `FeedsLengthMismatch` is raised; if the verified payload also has zero feeds, `NoFeeds()` is raised.

☐ Test coverage

- If the deadline has expired, the function reverts at the deadline check.

☒ Test coverage

- If the `calldata` is too short or malformed, the function may revert in the assembly decode step or within the `_verifyAndStore` path.

☐ Test coverage

- If `_verifyAndStore` fails signature/proof/feed/update validation, the function reverts and oracle data is not updated.

☒ Test coverage

- The verification fee is paid from the contract balance. `msg.value` from the same call is included in that balance and there is no separate refund.

☐ Test coverage

- Zero-feed `calldata` is not processed as a successful empty update. Depending on the feed count relationship with the verified payload, the function reverts with either `FeedsLengthMismatch` or `NoFeeds()`.

☐ Test coverage

- A successful verification does not guarantee that fresh, usable data is stored for each

selected feed. Missing or stale data may be skipped, and an invalid price may be stored as a stall marker.

☐ Test coverage

- External calls flow through the inherited `_verifyAndStore` (via `LazerConsumer`). `_verifyAndStore` verifies the DON-signed payload via `pythLazer.verifyUpdate`, then decodes the returned payload with a strict parser and, for each selected feed, evaluates timestamp monotonicity (`isAfter`), missing data, and invalid-price conditions before deciding to store, skip, or write a stall marker. These branching behaviors are described in the Branches section above.

☐ Test coverage

Function call analysis

- `pythLazer.verifyUpdate{value: PYTH_VERIFICATION_FEE}(priceUpdate)` (via internal `_verifyAndStore` → `LazerConsumer`)
 - **What is controllable?** `priceUpdate` (the price update payload) is controlled by the caller. `pythLazer` (the verifier address) and `PYTH_VERIFICATION_FEE` are immutable/constant. The fee is paid from the contract balance, and `msg.value` is included in that balance (with no separate refund).
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned payload is treated as a blob that has passed DON verification and is decoded by a strict parser. An arbitrary payload that has not passed verification never reaches the decode step (because `verifyUpdate` reverts). A successful verification only means the overall payload is validated; individual selected-feed storage may be skipped or written as a stall marker depending on timestamp, missing-data, and invalid-price conditions.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If `verifyUpdate` reverts (due to signature/format mismatch, insufficient fee, etc.), the entire update reverts. `pythLazer` is assumed to be the trusted Pyth Lazer contract.

Function: `function acceptStateGuardRole(bytes32 feedId) external feedExists(feedId)`

`acceptStateGuardRole` is the acceptance step of a two-step privilege transfer, in which the pending state guard designated by `setStateGuardRole` calls this function directly to finalize acquisition of the state guard role for the given feed. Once the role is confirmed, the `checkRole`-protected service functions for that feed (`setBaseAndQuoteTokens`, `setPriceGuard`, `setStateGuardRole`, `purgePendingStateGuardRole`, `purgeStateGuardRole`) can only be called by this guard, rather than falling back to `ADMIN_ROLE`.

Inputs

- feedId
 - **Control:** Pending guard only (the value itself is arbitrary)
 - **Constraints:** Must be a registered feed (feedExists) and pendingStateGuard[feedId] == msg.sender must hold.
 - **Impact:** Determines which feed's state guard role is being accepted.

Branches and code coverage

Intended branches

- If pendingStateGuard[feedId] == msg.sender, the function deletes pendingStateGuard[feedId], sets stateGuard[feedId] to msg.sender, and emits StateGuardUpdated.
 - ☒ Test coverage
- After acceptance, checkRole(feedId) for that feed takes the stateGuard[feedId] != address(0) branch, so holders of ADMIN_ROLE can no longer call the service functions for that feed — only the guard itself can.
 - ☒ Test coverage

Negative branches

- If the feed does not exist, the feedExists modifier reverts with FeedNotFound(feedId).
 - ☐ Test coverage
- If pendingStateGuard[feedId] != msg.sender (including the case where it is unset and equals the zero address), the function reverts with InvalidGuard(msg.sender) and the role is not transferred.
 - ☒ Test coverage
- Once stateGuard[feedId] is set, purgeStateGuardRole and setStateGuardRole are also protected by checkRole(feedId) and can only be called by the guard, meaning ADMIN_ROLE cannot reclaim or replace the state guard for that feed on its own. If the guard address is incorrect or the key is lost, state management for that feed may be permanently locked to that address.
 - ☐ Test coverage

Function: `function price(bytes32 feedId, address pool) external feedExists(feedId) notBlacklisted returns (uint256 mid, uint256 spread, uint16 volatility, uint256 refTime)`

`price` is the abuse-protected pool read path inherited from `OracleBase`. The caller must be the pool's price provider, the supplied pool must report that provider as being in-swap, and the function must pass feed/pool registration and blacklist checks.

Inputs

- `feedId`
 - **Control:** Arbitrary
 - **Constraints:** Must be a registered feed and `registeredPool[feedId][pool]` must be true.
 - **Impact:** Determines which oracle data feed is read.
- `pool`
 - **Control:** Arbitrary, but constrained by pool/provider binding
 - **Constraints:** Must not be the zero address, `pool.inSwap() == msg.sender` must hold, and the pool must not be blacklisted.
 - **Impact:** The pool to which the price read is attributed; subject to registration and blacklist checks.

Branches and code coverage

Intended branches

- If the feed exists and the caller/pool binding, registration, and blacklist checks all pass, the function returns raw oracle data and records a read event. The oracle path itself does not enforce staleness, price guard, or positive-price/stall-marker validation; those validations are the responsibility of the provider/consumer.

☒ Test coverage

Negative branches

- If the feed does not exist or the caller is blacklisted, the function reverts.

☒ Test coverage

- If the pool is the zero address or the `inSwap()` binding does not match, the function reverts.

☒ Test coverage

- If the pool is blacklisted or is not registered for the given feed, the function reverts.

☒ Test coverage

- The return value is raw oracle data from the gated read path. This function itself does not enforce freshness, price guard, or stall-marker exclusion.

☐ Test coverage

Function call analysis

- `IPool(pool).inSwap()`
 - **What is controllable?** The caller provides the `pool` address.
 - **If the return value is controllable, how is it used and how can it go wrong?** The return value must match `msg.sender` for the price read to be permitted.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the price read fails.

Function: `function integratorPrice(bytes32 feedId) external feedExists(feedId) notBlacklisted returns (uint256 mid, uint256 spread, uint16 volatility, uint256 refTime)`

`integratorPrice` is the whitelisted integrator read path inherited from `OracleBase`. Only callers included in the integrator allowlist can read the feed price, without requiring pool registration.

Inputs

- `feedId`
 - **Control:** Arbitrary
 - **Constraints:** Must be a registered feed.
 - **Impact:** Determines which oracle data feed is read.

Branches and code coverage

Intended branches

- If the feed exists and the caller is in the integrator allowlist, the function returns raw oracle data and records a read event. This function itself does not perform staleness, price guard, or positive-price/stall-marker validation.

☒ Test coverage

Negative branches

- If the feed does not exist or the caller is blacklisted, the function reverts.

- ☒ Test coverage
 - If the caller is not in the integrator allowlist, the function reverts.
- ☒ Test coverage
 - An integrator read returns raw oracle data. Provider-level freshness, price guard, and stall-marker/positive-price validation do not apply to integrator reads that bypass provider logic.
- ☐ Test coverage

Function: `function register(bytes32 feedId, address pool, address factory) external payable feedExists(feedId)`

`register` is the permissionless paid pool registration function inherited from `OracleBase`. An approved factory must recognize the pool as valid, and paying at least the registration fee activates the feed/pool read permission.

Inputs

- `feedId`
 - **Control:** Arbitrary
 - **Constraints:** Must be a registered feed.
 - **Impact:** Determines which feed the pool is registered under.
- `pool`
 - **Control:** Arbitrary
 - **Constraints:** Must not be the zero address and `factory.isPool(pool)` must return true.
 - **Impact:** The pool that will be permitted in the subsequent price read path.
- `factory`
 - **Control:** Arbitrary
 - **Constraints:** Must be included in the approved factory set.
 - **Impact:** The reference factory used to validate pool validity.
- `msg.value`
 - **Control:** Arbitrary
 - **Constraints:** Must be at least `registrationFee`.
 - **Impact:** If insufficient, registration fails; any excess is not refunded.

Branches and code coverage

Intended branches

- If the value is sufficient, the feed exists, the factory is approved, and the pool is valid, the function sets `registeredPool[feedId][pool]` to true.

☒ Test coverage

- If the pool was previously blacklisted, a successful registration clears it from the blacklist.

☒ Test coverage

Negative branches

- If the value is insufficient or the feed does not exist, the function reverts.

☒ Test coverage

- If the pool is the zero address or the factory is not in the approved set, the function reverts.

☒ Test coverage

- If `factory.isPool(pool)` returns false or reverts, registration fails.

☒ Test coverage

Function call analysis

- `IPoolFactory(factory).isPool(pool)`
 - **What is controllable?** The caller provides both `factory` and `pool`.
 - **If the return value is controllable, how is it used and how can it go wrong?** The approved factory must return true for pool registration to be permitted.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire registration fails.

Function: `function addApprovedFactory(address factory) external onlyRole(ADMIN_ROLE)`

`addApprovedFactory` is an `OracleBase` admin function inherited by this contract that adds a factory to the approved set, enabling it to be trusted in pool registration.

Inputs

- `factory`

- **Control:** ADMIN_ROLE Only
- **Constraints:** Must not be the zero address and must not already be in the approved set.
- **Impact:** Subsequent calls to register can trust the isPool(pool) result from this factory.

Branches and code coverage

Intended branches

- If the normal conditions are met, the function performs the intended state change or returns the expected read result.

☒ Test coverage

Negative branches

- If the permission, existence, or input validity conditions are not met, the function reverts.

☒ Test coverage

Function: function removeApprovedFactory(address factory) external onlyRole(ADMIN_ROLE)

removeApprovedFactory is an OracleBase admin function inherited by this contract that removes a factory from the approved factory set.

Inputs

- factory
 - **Control:** ADMIN_ROLE Only
 - **Constraints:** Must currently be in the approved set.
 - **Impact:** After removal, the factory can no longer be used for new pool registrations. The registeredPool state of already-registered pools is not changed by this function.

Branches and code coverage

Intended branches

- If the caller is an admin and the factory is in the approved set, the function removes it

from the approved set. Existing pool registration states are not modified.

☒ Test coverage

Negative branches

- If the caller is not an admin or the factory is not in the approved set, the function reverts.

☒ Test coverage

Function: `function setBlacklist(address account, bool value) external onlyRole(ADMIN_ROLE)`

`setBlacklist` is an OracleBase admin function inherited by this contract that sets the read-access blacklist status of an account. The blacklist applies to callers and pools in the price read path, as well as callers of `integratorPrice`. A pool's blacklist status can be reverted to false through a successful `register` call.

Inputs

- `account`
 - **Control:** ADMIN_ROLE Only
 - **Constraints:** Must not be the zero address.
 - **Impact:** The address whose blacklist status will be changed.
- `value`
 - **Control:** ADMIN_ROLE Only
 - **Constraints:** A boolean flag.
 - **Impact:** If true, blocks the applicable read paths for the account; if false, unblocks them.

Branches and code coverage

Intended branches

- If the caller is an admin and the account is nonzero, the function sets the blacklist value. If the value is already the same, it returns without modifying state.

☒ Test coverage

Negative branches

- If the caller is not an admin or the account is zero, the function reverts. A pool account's blacklist status of true can be changed to false by a subsequent successful `register`

call.

☒ Test coverage

Function: `function withdrawEth() external onlyRole(ADMIN_ROLE)`

`withdrawEth` is a function by which the `ADMIN_ROLE` holder withdraws the contract's entire ETH balance — including accumulated registration fees, overpayments, and any other ETH received — to the caller. Sweeping the entire balance without a partial-withdrawal option is intentional design.

Inputs

- No arguments
 - **Control:** `ADMIN_ROLE` only
 - **Constraints:** The caller must hold `ADMIN_ROLE`.
 - **Impact:** The recipient of the contract's entire ETH balance is always `msg.sender`.

Branches and code coverage

Intended branches

- If the caller holds `ADMIN_ROLE`, the function transfers the full `address(this).balance` to `msg.sender` via `call` and emits `EthWithdrawn(msg.sender, amount)`. If the balance is 0, a successful transfer of `amount == 0` completes normally.

☒ Test coverage

Negative branches

- If the caller does not have `ADMIN_ROLE`, the `OpenZeppelin onlyRole(ADMIN_ROLE)` check reverts and the balance does not move.
- If the ETH transfer (`call`) to `msg.sender` fails, the function reverts at `require(ok)`. If the receiving admin is a contract that rejects ETH, the withdrawal may be blocked.
- The recipient is always `msg.sender` rather than a separate destination argument, and the entire balance is swept, so any admin can unilaterally withdraw the entire balance to themselves when multiple admins exist.

☐ Test coverage

Function call analysis

- `payable(msg.sender).call{value: amount}("")`
 - **What is controllable?** The recipient is fixed to `msg.sender` (ADMIN), and `amount` is determined by the contract's total balance.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned success flag is validated by `require(ok)`, which reverts the entire call on failure.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If the admin is an untrusted contract, it may attempt reentrancy during the transfer; however, because the balance has already been withdrawn, there is no remaining amount to sweep on subsequent calls, and the function has no other state changes, so the additional effect is limited.

5. Assessment Results

During our assessment on the scoped Metric OMM contracts, we discovered five findings. No critical issues were found. Three findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.