Covenant Protocol — Technical Report

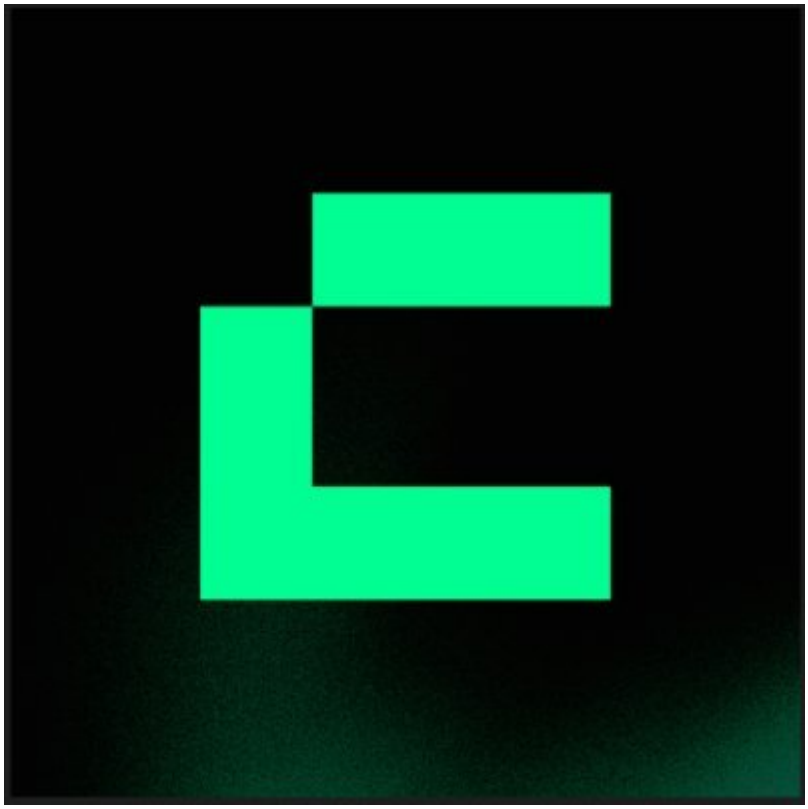
Audit: Code4rena — October 2025

Codebase: `2025-10-covenant`

Solidity Version: 0.8.30

EVM Target: Cancun

In-Scope Lines (nSLoC): ~2,281



Audited by: rubencrxz

Table of Contents

---

1. Protocol Overview

Covenant is a **permissionless structured-products protocol** deployed on EVM-compatible chains. It allows users to deposit a base (collateral) ERC20 token into isolated markets and receive two synthetic ERC20 tokens in return:

- **aToken** (leverage token): represents a leveraged long position on the base asset.
- **zToken** (debt/yield token): represents the debt leg of the position, with an embedded perpetual interest rate.

The protocol integrates an internal AMM called **LatentSwap**, a concentrated-liquidity exchange model inspired by Uniswap V3, which governs the relative pricing between aToken and zToken. Each market is isolated and identified by a deterministic ID derived from its four defining parameters: base token, quote token, oracle curator, and exchange model.

The primary operations exposed to end users are:

Operation	Description
`mint`	Deposit base token → receive aToken + zToken
`redeem`	Burn aToken + zToken → receive base token
`swap`	Exchange any asset pair within a market (BASE ↔ aToken, BASE ↔ zToken, aToken ↔ zToken)
`updateState`	Advance market state (interest accrual, fee accumulation) without a user operation
`multicall`	Atomic batch of the above operations

---

2. Repository Structure

```
src/
├── Covenant.sol                    # Main protocol entry point (274 nSLoC)
├── interfaces/
│   ├── ICovenant.sol               # Core protocol interface
│   ├── ILiquidExchangeModel.sol    # LEX interface
│   ├── IPriceOracle.sol            # Oracle interface
│   └── ISynthToken.sol             # Synthetic token interface
├── libraries/
│   ├── Errors.sol                  # Protocol-level error definitions
│   ├── Events.sol                  # All protocol events (50 nSLoC)
│   ├── MarketParams.sol            # Market ID derivation (10 nSLoC)
│   ├── ValidationLogic.sol         # Input validation library (74 nSLoC)
│   ├── MultiCall.sol               # Multicall implementation (8 nSLoC)
```

```
│   ├── NoDelegateCall.sol           # Delegatecall prevention (12 nSLoC)
│   ├── SafeMetadata.sol             # Tolerant ERC20 metadata reads (47 nSLoC)
│   └── Utils.sol                    # General utilities (10 nSLoC)
├── curators/
│   ├── CovenantCurator.sol          # Oracle router (91 nSLoC)
│   ├── lib/Errors.sol               # Curator error definitions
│   ├── interfaces/ICovenantPriceOracle.sol
│   └── oracles/
│       ├── BaseAdapter.sol          # Abstract oracle adapter (28 nSLoC)
│       ├── CrossAdapter.sol         # Cross-price oracle chaining (64 nSLoC)
│       ├── chainlink/ChainlinkOracle.sol   # Chainlink push feeds (26 nSLoC)
│       └── pyth/PythOracle.sol             # Pyth pull feeds (63 nSLoC)
├── lex/latentswap/
│   ├── LatentSwapLEX.sol            # Exchange model implementation (310 nSLoC)
│   ├── interfaces/
│   │   ├── ILatentSwapLEX.sol
│   │   └── ITokenData.sol
│   └── libraries/
│       ├── LatentSwapLogic.sol      # Core DEX mathematics (656 nSLoC)
│       ├── LatentMath.sol           # Protocol-specific math (159 nSLoC)
│       ├── DebtMath.sol             # Perpetual debt calculations (36 nSLoC)
│       ├── FixedPoint.sol           # Fixed-point constants (17 nSLoC)
│       ├── SqrtPriceMath.sol        # Square-root price math (79 nSLoC)
│       ├── SaturatingMath.sol       # Overflow-safe arithmetic (34 nSLoC)
│       ├── TokenData.sol            # Token metadata overrides (65 nSLoC)
│       ├── Uint512.sol              # 512-bit integer arithmetic (66 nSLoC)
│       └── LSErrors.sol             # LEX error definitions (23 nSLoC)
├── synths/
│   └── SynthToken.sol               # Synthetic ERC20 (48 nSLoC)
└── periphery/
    ├── DataProvider.sol             # View aggregator for frontends
    ├── interfaces/IDataProvider.sol
    └── libraries/LatentSwapLib.sol
```

Build configuration:

- Optimizer runs: 3,750
- Fuzz iterations (testing): 10,000

---

3. External Dependencies

Alias	Library	Usage
`@openzeppelin`	OpenZeppelin Contracts	ERC20, ERC20Metadata, Ownable2Step, SafeERC20, IERC4626, Math, SafeCast
`@solady`	Solady	FixedPointMathLib (ln, exp, wadLn)
`@euler-price-oracle`	Euler Price Oracle	BaseAdapter, ChainlinkOracle, PythOracle, ScaleUtils

Alias	Library	Usage
`@chainlink`	Chainlink Brownie Contracts	AggregatorV3Interface
`@aave`	Aave V3 Core	PercentageMath (basis-point arithmetic)
`@clones`	Clones with Immutable Args	Clone factory patterns
`@std`	Forge Std	Test utilities (not in scope)

The Pyth Network SDK is used as an interface dependency for pull-oracle price updates (`IPyth`, `PythStructs`).

---

4. Contract Inventory

4.1 Covenant.sol

The singleton entry point for all protocol operations.

- **Inheritance:** `ICovenant`, `NoDelegateCall`, `Ownable2Step`
- **State:**
  - `mapping(MarketId => MarketState) markets` — per-market operational state
  - `mapping(address => bool) enabledLex` — whitelisted LEX implementations
  - `mapping(address => bool) enabledCurators` — whitelisted oracle routers
  - `uint32 defaultProtocolFee` — base fee applied to new markets
  - `address defaultPauseAddress` — default emergency address
- **Key modifiers:**
  - `lock(marketId)` — per-market reentrancy lock; verifies market is UNLOCKED before each call, sets to LOCKED during execution
  - `lockView(marketId)` — read access; allows PAUSED markets
  - `noDelegateCall` — inherited; rejects delegatecall context

4.2 LatentSwapLEX.sol

The Liquid Exchange Model (LEX). Implements the LatentSwap concentrated-liquidity DEX logic for a single LEX deployment shared across all markets that reference it.

- **Inheritance:** `ILatentSwapLEX`, `TokenData`, `Ownable2Step`
- **State:**
  - `mapping(MarketId => LexState) lexStates`
  - `mapping(MarketId => LexConfig) lexConfigs`
  - `mapping(MarketId => SynthTokens) synthTokens`
  - Per-market `LexParams` (immutable at init, stored at market creation)
- **Access:** All mint/redeem/swap/updateState calls are restricted to the Covenant core via `onlyCovenantCore()`.

4.3 SynthToken.sol

One contract instance per synthetic token (two per market: aToken and zToken).

- **Inheritance:** `ERC20`, `ISynthToken`
- **Immutable fields:** `_covenantCore`, `_lexCore`, `_marketId`, `_synthType`, `_decimals`
- **Restricted functions:** `lexMint()` and `lexBurn()` only callable by the associated `_lexCore` address.

4.4 CovenantCurator.sol

Oracle router that resolves asset pair prices to a configured oracle implementation.

- **Inheritance:** `Ownable2Step`, `IPriceOracle`
- **State:**
  - `mapping(bytes32 => address) oracles` — keyed by lexicographically sorted `(assetA, assetB)` pair hash
  - `mapping(address => address) resolvedVaults` — ERC4626 vault → underlying asset routing
  - `address fallbackOracle` — fallback when no direct oracle is configured

4.5 Oracle Adapters

Contract	Base	Feed Type	Update Model
`ChainlinkOracle`	`EulerChainlinkOracle`	AggregatorV3	Push (no fee)
`PythOracle`	`EulerPythOracle`	Pyth price feeds	Pull (user-paid fee)
`CrossAdapter`	`BaseAdapter`	Derived (two oracles)	Delegated to both legs

4.6 Libraries

Library	Purpose
`LatentSwapLogic`	Core mint/redeem/swap computations (656 nSLoC)
`LatentMath`	Protocol-specific math: ETWAP, debt interest
`DebtMath`	Perpetual debt notional price update
`SqrtPriceMath`	Uniswap V3-style square-root price arithmetic
`SaturatingMath`	Saturating add/sub/mul (no revert on overflow)
`Uint512`	512-bit integer support for high-precision multiplication
`ValidationLogic`	Input validation for all user-facing parameters
`MarketParams`	`toId()` — derives `MarketId` as `bytes20(keccak256(abi.encode(params)))`

5. Data Structures and Storage Layout

5.1 Market Identification

```
type MarketId is bytes20;

struct MarketParams {
    address baseToken;   // Collateral asset
    address quoteToken;  // Pricing reference asset
    address curator;     // Oracle router
    address lex;         // Exchange model
}
```

`MarketId = bytes20(keccak256(abi.encode(MarketParams)))` — deterministic, collision-resistant identifier.

5.2 Market State (Covenant.sol)

```
struct MarketState {
    uint256 baseSupply;                // Total base tokens held in this market
    uint128 protocolFeeGrowth;         // Cumulative uncollected protocol fees
    address authorizedPauseAddress;    // Per-market emergency address
    uint8   statusFlag;                // 0=uninitialized, 1=unlocked, 2=locked,
3=paused
}
```

5.3 LEX State (LatentSwapLEX.sol)

```
struct LexState {
    uint256 lastDebtNotionalPrice;  // Intrinsic value of zToken in WAD
    uint256 lastBaseTokenPrice;     // Oracle price of base token in WAD
    uint256 lastETWAPBaseSupply;    // Exponential time-weighted average of
baseSupply
    uint160 lastSqrtPriceX96;       // DEX price in Q96 sqrt format
    uint96  lastUpdateTimestamp;    // Last update block timestamp
    int64   lastLnRateBias;         // Log-rate bias term in WAD
}

struct LexConfig {
    uint32  protocolFee;   // Encoded: (yieldFee << 16) | tvlFee in BPS
    address aToken;        // Leverage token address
    address zToken;        // Debt token address
    uint8   noCapLimit;    // Power-of-2 threshold for mint/redeem caps
    int8    scaleDecimals; // quoteDecimals - synthDecimals
    bool    adaptive;      // Whether debt pricing uses adaptive mode
}
```

5.4 LEX Initialization Parameters

```
struct LexParams {
    address covenantCore;
    int64   initLnRateBias;          // Initial log-rate bias
    uint160 edgeSqrtPriceX96_B;      // Upper price boundary (edge B)
    uint160 edgeSqrtPriceX96_A;      // Lower price boundary (edge A)
    uint160 limHighSqrtPriceX96;     // Max LTV limit price
    uint160 limMaxSqrtPriceX96;      // Absolute maximum price
    uint32  debtDuration;            // Perpetual debt period in seconds
    uint8   swapFee;                 // DEX swap fee in BPS (0–255)
    uint256 targetXvsL;              // Pre-computed target liquidity ratio
}
```

5.5 Synthetic Token Enumeration

```
enum AssetType { BASE, DEBT, LEVERAGE, COUNT }

struct SynthTokens {
    address aToken;   // Leverage token (LEVERAGE)
    address zToken;   // Debt token (DEBT)
}
```

5.6 Operation Parameter Structs

```
struct MintParams {
    MarketId   marketId;
    MarketParams marketParams;
    uint256    baseAmountIn;
    address    to;
    uint256    minATokenAmountOut;
    uint256    minZTokenAmountOut;
    bytes      data;        // Oracle update payload
    uint256    msgValue;    // ETH for oracle fees
}

struct RedeemParams {
    MarketId   marketId;
    MarketParams marketParams;
    uint256    aTokenAmountIn;
    uint256    zTokenAmountIn;
    address    to;
    uint256    minAmountOut;
    bytes      data;
    uint256    msgValue;
}

struct SwapParams {
    MarketId   marketId;
    MarketParams marketParams;
```

```
    AssetType  assetIn;
    AssetType  assetOut;
    address    to;
    uint256    amountSpecified;
    uint256    amountLimit;   // minOut (exact-in) or maxIn (exact-out)
    bool       isExactIn;
    bytes      data;
    uint256    msgValue;
}
```

5.7 Internal Market State Cache

A transient struct computed at the start of each operation:

```
struct LexFullState {
    LexState   lexState;
    LexConfig  lexConfig;
    uint256    baseTokenSupply;
    uint256[2] supplyAmounts;       // [zToken supply, aToken supply]
    uint256[2] dexAmounts;          // [debt DEX value, leverage DEX value]
    uint256[3] dexAmountsScaled;    // Decimal-adjusted values
    uint256[3] synthAmountsScaled;
    uint256    liquidityRatioX96;   // price × liquidity ratio
    uint160    liquidity;           // Uniswap V3-style L
    uint96     accruedProtocolFee;
    bool       underCollateralized;
}
```

---

6. Core Protocol Flows

6.1 Market Creation (`createMarket`)

1. `ValidationLogic.checkMarketParams()` verifies LEX and curator are enabled and the market does not yet exist.
2. `LatentSwapLEX.initMarket()` is called:
   - Deploys two `SynthToken` instances (aToken and zToken).
   - Reads the initial oracle price from the curator.
   - Initializes `LexState.lastSqrtPriceX96` at target LTV (Q96 = 1:1 ratio).
   - Stores `LexConfig` and `LexParams`.
3. `Covenant` sets `markets[id].statusFlag = STATE_UNLOCKED`.
4. Emits `CreateMarket` event with full market parameters.

6.2 Mint Flow

```
User → Covenant.mint(MintParams)
          │
          ├─ lock(marketId)
```

```
          ├─ ValidationLogic.checkMintParams()
          ├─ LatentSwapLEX.mint()
          │     ├─ _updateOraclePrice()      ← curator.updatePriceFeeds() if data
present
          │     ├─ _calcBasePrice()          ← curator.getQuote()
          │     ├─ _calculateMarketState()   ← accrues interest, updates ETWAP
          │     ├─ LatentSwapLogic.mintLogic()
          │     │     ├─ Checks mint cap (ETWAP-based)
          │     │     ├─ Calculates aToken + zToken outputs
          │     │     └─ Validates liquidity invariant
          │     ├─ SynthToken(aToken).lexMint(to, aAmount)
          │     └─ SynthToken(zToken).lexMint(to, zAmount)
          ├─ Covenant.marketState.baseSupply += baseAmountIn − protocolFee
          ├─ SafeERC20.safeTransferFrom(baseToken, user → Covenant)
          └─ unlock(marketId)
```

Outputs: `aTokenAmountOut`, `zTokenAmountOut` — both subject to slippage parameters `minATokenAmountOut` and `minZTokenAmountOut`.

6.3 Redeem Flow

```
User → Covenant.redeem(RedeemParams)
          │
          ├─ lock(marketId)
          ├─ ValidationLogic.checkRedeemParams()
          ├─ LatentSwapLEX.redeem()
          │     ├─ _updateOraclePrice() / _calcBasePrice()
          │     ├─ _calculateMarketState()
          │     ├─ Optional internal rebalance (internal swap)
          │     ├─ LatentSwapLogic.redeemLogic()
          │     │     ├─ Checks redeem cap (ETWAP-based)
          │     │     ├─ Calculates baseAmountOut
          │     │     └─ Validates liquidity invariant
          │     ├─ SynthToken(aToken).lexBurn(user, aAmount)
          │     └─ SynthToken(zToken).lexBurn(user, zAmount)
          ├─ Validates baseAmountOut >= minAmountOut
          ├─ Validates baseSupply is sufficient
          ├─ Covenant.marketState.baseSupply -= baseAmountOut + protocolFee
          ├─ SafeERC20.safeTransfer(baseToken, Covenant → user)
          └─ unlock(marketId)
```

6.4 Swap Flow

Swaps operate between any two of the three asset types (BASE, DEBT, LEVERAGE).

```
User → Covenant.swap(SwapParams)
          │
          ├─ lock(marketId)
          ├─ ValidationLogic.checkSwapParams()
```

```
        ├─ LatentSwapLEX.swap()
        │     ├─ _updateOraclePrice() / _calcBasePrice()
        │     ├─ _calculateMarketState()
        │     ├─ LatentSwapLogic.swapLogic()
        │     │     ├─ Applies swap fee
        │     │     ├─ Computes amountOut / amountIn via invariant
        │     │     └─ Updates lastSqrtPriceX96
        │     ├─ Burns input synth(s) / mints output synth(s)
        │     └─ Returns base delta if BASE is involved
        ├─ If BASE in swap: update baseSupply, transfer base token
        └─ unlock(marketId)
```

Exact-in / Exact-out: Both modes are supported via `isExactIn`. Slippage is enforced via `amountLimit`.

6.5 State Update (`updateState`)

A permissionless call that advances the market state (interest accrual, ETWAP update, fee accumulation) without minting or redeeming. Can be called by any external actor (keeper, user, MEV bot).

6.6 Protocol Fee Collection (`collectProtocolFee`)

Owner-restricted. Reads `marketState.protocolFeeGrowth`, resets it to zero, and transfers the accumulated base tokens from `Covenant` balance to the recipient.

---

7. Mathematical Model

7.1 Fixed-Point Precision Systems

System	Scale	Value of 1.0	Usage
WAD	18 decimals	`1e18`	Token prices, debt notional, general amounts
Q96	96-bit	`2^96`	sqrtPrice in DEX calculations (Uniswap V3 compatible)
RAY	27 decimals	`1e27`	Interest rate compounding

Conversions between systems are handled by `SafeCast`, `Math.mulDiv`, and Solady's `FixedPointMathLib`.

7.2 LatentSwap Concentrated Liquidity Invariant

The LatentSwap AMM uses a Uniswap V3-derived invariant adapted to the two-synth model:

```
(L / sqrtP_A  −  V_debt) × (L × sqrtP_B  −  V_leverage)  =  L²

Where:
  L         = liquidity invariant (uint160, Uniswap V3 compatible)
  sqrtP_A   = lower price edge (edgeSqrtPriceX96_A)
  sqrtP_B   = upper price edge (edgeSqrtPriceX96_B)
  V_debt    = total debt token value (scaled to quote decimals)
  V_leverage = total leverage token value (scaled to quote decimals)
```

At market initialization, `sqrtPrice = Q96` places the market at a 1:1 price ratio (target LTV). Price movement towards `sqrtP_B` represents increasing leverage; towards `sqrtP_A` represents debt dominance.

7.3 Perpetual Debt Interest Model

The zToken (debt token) accrues interest continuously via a variable rate:

1. **Debt notional price** `D_t` starts at `WAD` (1.0) and decreases over time as interest accrues.
2. The rate is driven by:
   - The **log-rate bias** `lnRateBias` (market-specific, set at init)
   - The **debt discount**: how far the current DEX price is from par (1.0 WAD)
   - The **debt duration** `debtDuration` (seconds per perpetual period)
3. Update formula (simplified):

   ```
   D_new = D_old × exp(−rate × Δt / debtDuration)
   ```

   where `rate` is derived from the current log-rate bias and market conditions.
4. In **adaptive mode** (`LexConfig.adaptive = true`), the rate also adjusts based on the current market price deviation from target.
5. The `DebtMath` library computes the update with Taylor series approximation for gas efficiency.

7.4 Exponential Time-Weighted Average of Base Supply (ETWAP)

Used to compute mint/redeem caps:

```
ETWAP_new = ETWAP_old × exp(−λ × Δt)  +  baseSupply_current × (1 − exp(−λ × Δt))
```

A rolling average with exponential decay. The half-life is implied by `λ` (hardcoded per-protocol). This average is compared against `2^noCapLimit` to determine whether size limits apply.

7.5 Mint and Redeem Caps

- **Condition:** Caps are inactive when `baseSupply < 2^noCapLimit`.
- **When active:**
  - `MAX_MINT_FACTOR_CAP`: limits single mint to ≤ 100% of current ETWAP per operation.
  - `MAX_REDEEM_FACTOR_CAP`: limits single redeem to ≤ 25% of current ETWAP per operation.
- **Purpose:** Limit price impact of large single operations relative to pool size.

7.6 Square-Root Price Arithmetic

`SqrtPriceMath.sol` implements:

- `getAmount0Delta`: Amount of token0 for a price range change.
- `getAmount1Delta`: Amount of token1 for a price range change.

- `getNextSqrtPriceFromInput` / `getNextSqrtPriceFromOutput`: New price after a swap of specified size.

All arithmetic uses `Uint512.sol` for intermediate 512-bit precision to avoid overflow in `(sqrtP_new × L)` computations.

7.7 Saturating Arithmetic

`SaturatingMath.sol` provides `safeAdd`, `safeSub`, `safeMul` that clamp to `uint256.max` / `0` rather than reverting. Used in fee accrual and interest accumulation paths to ensure the protocol never halts due to arithmetic overflow in non-critical accumulator variables.

---

8. Oracle Architecture

8.1 Price Oracle Interface

All oracle adapters implement:

```
interface ICovenantPriceOracle {
    function getQuote(uint256 inAmount, address base, address quote) external view
returns (uint256);
    function getQuotes(uint256 inAmount, address base, address quote) external
view returns (uint256 bid, uint256 ask);
    function previewGetQuote(uint256 inAmount, address base, address quote, bytes
calldata data) external view returns (uint256);
    function updatePriceFeeds(bytes calldata data) external payable;
    function getUpdateFee(bytes calldata data) external view returns (uint256);
}
```

8.2 CovenantCurator — Oracle Router

`CovenantCurator` is a stateful router that maps `(baseToken, quoteToken)` pairs to oracle implementations. Resolution order:

1. Look up `oracles[hash(sorted(base, quote))]` for a direct oracle.
2. If not found, check `resolvedVaults[base]` or `resolvedVaults[quote]` to recursively resolve via ERC4626 vault underlying.
3. If still not found, use `fallbackOracle`.
4. If no fallback, revert.

For ERC4626 vault routing: `price(vault, quote) = vault.convertToAssets(1e18) × price(underlying, quote) / 1e18`.

8.3 Chainlink Oracle

- Wraps `EulerChainlinkOracle` from the Euler oracle library.
- Reads `AggregatorV3Interface.latestRoundData()`.
- Enforces `maxStaleness` (set at deployment time) against `block.timestamp − updatedAt`.

- Returns price scaled to 18 decimals using Euler's `ScaleUtils`.
- No on-chain update fee (Chainlink nodes push updates).

8.4 Pyth Oracle

- Wraps `EulerPythOracle` from the Euler oracle library.
- Requires caller to supply signed price update payloads (`bytes data`).
- Validates:
  - Confidence interval: `conf / abs(price) <= maxConfWidth` (BPS).
  - Price exponent: within acceptable bounds.
  - Staleness: `block.timestamp - publishTime <= maxStaleness`.
- `getUpdateFee(data)` queries the Pyth contract for fee before user submits.
- ETH fee passed via `msgValue` in operation structs, forwarded through Covenant → LEX → Curator → PythOracle → IPyth.

8.5 CrossAdapter

Chains two oracle lookups for indirect pricing:

```
price(base, quote) = price(base, cross) × price(cross, quote)
```

- Supports bidirectional resolution (inverse of a pair).
- `getUpdateFee` aggregates fees from both legs.
- `updatePriceFeeds` calls both legs.

8.6 Oracle Price Representation

All prices returned by oracles are in **WAD units**:

- `getQuote(1e18, baseToken, quoteToken)` returns the number of `quoteToken` units (in WAD) equivalent to `1e18` baseToken units.
- Internally, `LatentSwapLogic` uses the base price to calculate `liquidityRatioX96` and `sqrtPriceX96`.

---

9. Token Standards and Synthetic Tokens

9.1 Base Token

Any ERC20-compatible token. The protocol uses `SafeERC20` for all transfers. Tokens with non-standard `decimals()` or missing `symbol()`/`name()` are tolerated via `SafeMetadata.sol` (try/catch on metadata reads).

9.2 Quote Token

Reference asset for pricing. Does not need to be an ERC20 — its address is used only for oracle lookups. Metadata (decimals, symbol, name) can be overridden in `TokenData` if the asset is not ERC20-compliant (e.g., native ETH via ERC-7535).

9.3 SynthToken (aToken / zToken)

- Implements standard OpenZeppelin `ERC20`.
- One pair deployed per market at creation time.
- **Immutable state set at deployment:**
  - `_covenantCore`: Covenant contract address
  - `_lexCore`: LatentSwapLEX contract address
  - `_marketId`: Associated market ID
  - `_synthType`: `0 = aToken (LEVERAGE)`, `1 = zToken (DEBT)`
  - `_decimals`: Token decimals (matches base token decimals)
- `lexMint(address, uint256)` and `lexBurn(address, uint256)` are the only non-standard functions, restricted to `_lexCore`.
- No admin functions; the token is fully controlled by the LEX contract.

9.4 ERC4626 Support

ERC4626 vault shares can be used as intermediate assets in oracle routing via `CovenantCurator.resolvedVaults`. The vault's `convertToAssets()` function is used for price conversion; vault shares are not directly tradeable in the protocol.

---

10. Access Control and Permissions

10.1 Role Hierarchy

Role	Contract	Assigned To	Capabilities
Covenant Owner	`Covenant`	Governance (Ownable2Step)	Enable/disable LEX, enable/disable curators, set default protocol fee, set default pause address, collect fees
Market Pause Address	`Covenant`	Per-market (set by owner)	Pause/unpause market, transfer pause permission to another address
LEX Owner	`LatentSwapLEX`	Governance (Ownable2Step)	Set default mint/redeem cap, override token metadata
Curator Owner	`CovenantCurator`	Governance (Ownable2Step)	Configure oracle pairs, set fallback oracle, configure ERC4626 vault routing

10.2 Market Status Flags

```
uint8 constant STATE_UNINITIALIZED = 0;  // Does not exist
uint8 constant STATE_UNLOCKED      = 1;  // Normal operation
uint8 constant STATE_LOCKED        = 2;  // Reentrancy guard active
uint8 constant STATE_PAUSED        = 3;  // Market halted
```

The `lock(marketId)` modifier: verifies `statusFlag == UNLOCKED`, sets to `LOCKED`, executes, then resets to `UNLOCKED`. This is a per-market lock, so different markets can operate concurrently.

10.3 Two-Step Ownership

All governance contracts (`Covenant`, `LatentSwapLEX`, `CovenantCurator`) inherit `Ownable2Step` from OpenZeppelin, requiring a pending owner to accept the transfer before it takes effect.

10.4 Delegation Prevention

`NoDelegateCall.sol` records `address(this)` at deployment time and checks that it matches on each call. This prevents any external contract from using `delegatecall` to execute Covenant logic in a different storage context.

---

11. Fee Model

11.1 Fee Encoding

Protocol fees are encoded in a single `uint32` field:

```
protocolFee = (yieldFee << 16) | tvlFee
```

Component	Bits	Denomination	Maximum
`yieldFee`	upper 16	BPS of yield earned	3,000 BPS (30%)
`tvlFee`	lower 16	BPS of TVL per year	500 BPS (5%)

11.2 Fee Accrual

- Fees are computed inside `LatentSwapLogic` on every `mint`, `redeem`, `swap`, and `updateState` call.
- The computed fee is **immediately subtracted from `baseSupply`** in Covenant, so it does not affect the invariant calculations for future operations.
- The fee accumulates in `marketState.protocolFeeGrowth` (uint128).
- `SaturatingMath` ensures that the accumulator never overflows and reverts; it saturates instead.

11.3 Fee Collection

`Covenant.collectProtocolFee(marketId, recipient)` is owner-restricted:

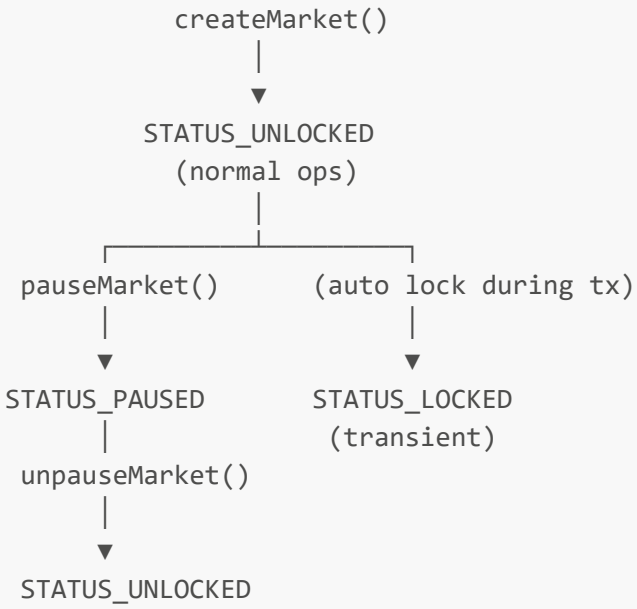
1. Reads `marketState.protocolFeeGrowth`.
2. Resets to zero.
3. Transfers `protocolFeeGrowth` in base tokens to `recipient`.

11.4 Swap Fee

A separate per-market swap fee (`LexParams.swapFee`, 0–255 BPS) applies to all DEX swaps. This fee increases the invariant (adds liquidity to the pool) rather than being extracted separately.

---

12. Market Lifecycle



Markets have **no destruction mechanism** — they cannot be deleted once created. A paused market blocks `mint`, `redeem`, and `swap` but allows read operations via `lockView`.

---

13. Multicall and Reentrancy Model

13.1 Multicall

`Covenant.multicall(bytes[] calldata calls)` enables atomic batches:

1. Records the contract's base token balance snapshot for each market.
2. Sets an internal `_isMulticall` flag.
3. Executes each encoded call via `address(this).call(calls[i])`.
4. After all calls complete, verifies the net balance change matches the sum of recorded deltas (no over- or under-deposit).

Intended use: Combine a Pyth price update payload with a mint in a single transaction.

13.2 Reentrancy Protection

Two mechanisms work in tandem:

1. **Per-market lock** (`statusFlag = LOCKED`): Prevents re-entry into the same market within a single call tree.
2. **Balance verification in multicall**: Ensures atomic accounting across batched calls.

The `NoDelegateCall` modifier prevents a third vector where a contract uses `delegatecall` to enter the Covenant logic in a different storage context.

---

14. Upgrade and Governance Model

14.1 No Upgradeable Proxies

The Covenant protocol does **not** use any proxy pattern (UUPS, Transparent, Beacon). All contract bytecode is immutable after deployment. There is no `implementation` address, no `upgradeTo()` function, and no storage collision risk from proxy layouts.

14.2 Governance-Controlled Mutable Surface

Flexibility is achieved through governance-controlled allowlists rather than upgrades:

Mutable Element	Controller	Mechanism
Enabled LEX implementations	Covenant Owner	`enableLex()` / `disableLex()`
Enabled oracle curators	Covenant Owner	`enableCurator()` / `disableCurator()`
Default protocol fee	Covenant Owner	`setDefaultProtocolFee()`
Per-market fee override	Covenant Owner	Set at market creation
Oracle pair configuration	Curator Owner	`CovenantCurator.setOracle()`
Fallback oracle	Curator Owner	`CovenantCurator.setFallbackOracle()`
ERC4626 vault routing	Curator Owner	`CovenantCurator.setResolvedVault()`
Token metadata overrides	LEX Owner	`TokenData` storage overrides
Mint/redeem cap threshold	LEX Owner	`noCapLimit` per market
Market pause state	Pause Address	`pauseMarket()` / `unpauseMarket()`

14.3 New Market Deployment Path

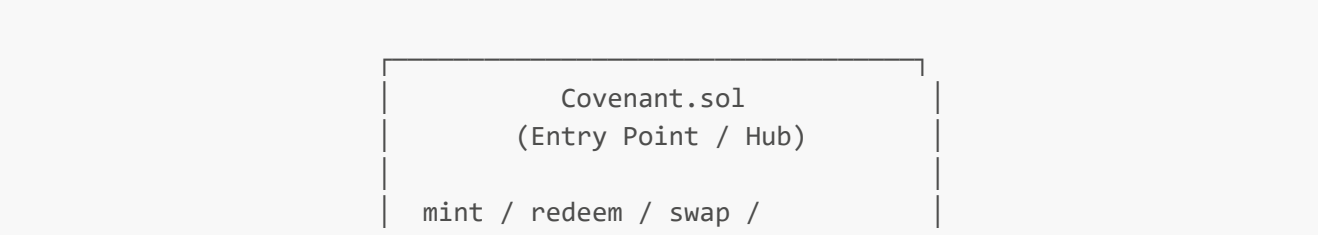
To add a new asset pair or LEX variant:

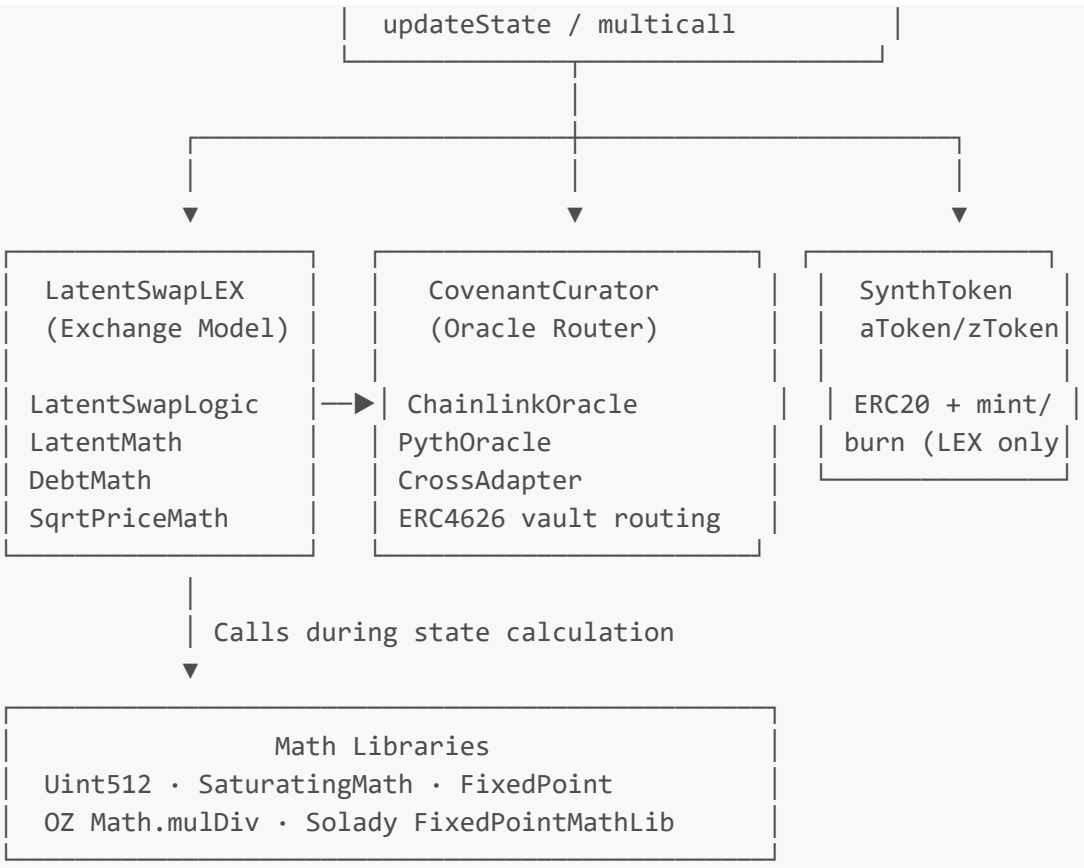
1. Deploy a new `LatentSwapLEX` implementation.
2. Owner enables it via `Covenant.enableLex(address)`.
3. Deploy or configure a `CovenantCurator` with the required oracle pairs.
4. Owner enables the curator via `Covenant.enableCurator(address)`.
5. Any permissionless actor calls `Covenant.createMarket(params, initData)`.

Existing markets are not affected by new deployments.

---

15. Contract Interaction Map





Data flow summary:

- `Covenant` is the sole user-facing contract; it holds all `baseToken` balances.
- `LatentSwapLEX` stores all market-level DEX state and controls synth minting/burning.
- `CovenantCurator` is stateless from the protocol's perspective — it routes oracle calls and returns prices.
- `SynthToken` contracts are passive ERC20 tokens; all logic is in `LatentSwapLEX`.
- No base tokens are ever held by `LatentSwapLEX` or `SynthToken`; the entire collateral pool resides in `Covenant`.

Findings

No valid High or Medium severity findings were identified in this audit.
