ChatGPT

FountFi Multi-Collateral BTC Vault Specification

This document specifies a new **FountFi Vault** implementation supporting **multi-collateral BTC derivative deposits**. It is a fork of the original FountFi RWA vault design [1], modified to accept multiple BTC-based assets and simplified to address prior stack-depth and complexity issues. The vault consolidates various wrapped BTC tokens into a unified pool, leveraging off-chain NAV reporting and a streamlined architecture.

1. Overview

The **Multi-Collateral BTC Vault** allows users to deposit any supported BTC-pegged token (e.g., WBTC, tBTC, cbBTC), and receive an ERC-4626 share token in return. The vault treats all supported BTC variants as equivalent in value to a base unit called **sovaBTC**, at a fixed 1:1 rate. Internally, vault accounting is denominated in sovaBTC units (8 decimal places each), while share tokens use 18 decimals for compatibility with DeFi standards [2]. On redemption, users always receive **sovaBTC**, regardless of which variant was originally deposited.

This design effectively creates a unified BTC clearing layer [3]: deposits of different BTC wrappers are pooled and interchangeable, enabling instant conversion between BTC wrapper types via the vault mechanism. Each sovaBTC redeemed is fully backed 1:1 by an equivalent amount of BTC or qualified BTC derivatives held in custody [4]. A trusted off-chain oracle (the **PriceOracleReporter**) periodically updates the vault's Net Asset Value (NAV) by reporting the price-per-share, ensuring accurate share valuation based on the underlying collateral [5].

Key improvements over the previous implementation include a **leaner contract architecture** (removing complex hooks, KYC gates, and nested logic) and the use of modular libraries for math and validation. This reduces stack depth and improves code maintainability, addressing prior "stack too deep" compiler issues [6] [7]. The vault operates permissionlessly by default – any user can deposit or redeem without pre-approval – while still allowing an admin/manager to manage collateral and liquidity.

2. User Flow

Deposit Flow: Users initiate a deposit by specifying one of the supported BTC derivative tokens and an amount. The deposit process is as follows:

- **Approval:** The user approves the vault (or its designated conduit contract) to transfer the specified BTC token on their behalf. This one-time approval covers all future deposits of that token [8].
- **Deposit Transaction:** The user calls the vault's deposit function (or a token-specific deposit method) indicating the token type and amount. The vault verifies the token is in its conversion registry of allowed BTC variants.
- **Asset Transfer:** The vault (or conduit) transfers the tokens from the user's address into the vault's custody [8]. All incoming assets are held by the vault's strategy contract and recorded in an internal balance mapping for each token type.

- **Conversion to Base Units:** The deposited amount is immediately normalized to **sovaBTC units** at a 1:1 rate. Since all variants share the same 8-decimal format, the conversion is value-equivalent (1 WBTC or 1 tBTC is treated as 1.00000000 sovaBTC). No exchange or price slippage occurs; the vault simply treats the deposit as if it were sovaBTC.
- **Share Minting:** The vault mints new ERC-4626 **share tokens** to the user as a receipt for their deposit. The number of shares is calculated based on the current price-per-share and deposit size in base units. Specifically, `sharesMinted = depositAmount_in_sovaBTC * (10^(18 - 8)) / (pricePerShare)` (adjusting for asset vs. share decimals) [2]. For example, if the price-per-share is 1.0 (initial NAV) and 1 WBTC is deposited, the user receives 1e18 shares (representing 1.00000000 BTC of value) [9]. If the NAV has grown (price-per-share > 1), the same deposit yields proportionally fewer shares.
- **Receipt:** The user's share token balance increases, and an event is emitted recording the deposit. The vault's total managed asset value (denominated in sovaBTC) increases by the deposit amount.

After deposit, users hold **ERC-4626 vault shares** which accrue value based on the vault's NAV updates. They do *not* directly receive sovaBTC at deposit time – sovaBTC is only distributed on redemption to maintain a clear separation between deposit receipts and the unified BTC token.

Redemption (Withdrawal) Flow: To redeem, a user returns their vault share tokens and receives **sovaBTC** in exchange:

- **Redeem Request:** The user calls `redeem(shares, to)` on the share contract (or an equivalent withdraw function specifying an amount of underlying to receive). The vault calculates the sovaBTC amount corresponding to the submitted shares using the current price-per-share (including any accrued yield or updates). For instance, if the user has 1.0 share and the current price-per-share is 1.1 sovaBTC, the redemption yields 1.1 sovaBTC (in 8 decimals) [10].
- **Liquidity Check:** Before transferring assets out, the vault checks its available sovaBTC liquidity. **Redemptions are only possible if the vault (or its manager) has supplied enough sovaBTC liquidity to fulfill the request.** In practice, the vault's strategy contract must hold at least the equivalent sovaBTC for the redemption. This sovaBTC liquidity is provided by the admin/manager ahead of time (see Section 4), ensuring that redemptions are fully backed. If sufficient sovaBTC is on hand, the redemption proceeds; otherwise, the transaction will revert or queue (depending on configuration), since the vault cannot issue sovaBTC on its own.
- **Asset Transfer:** The vault burns the user's shares and transfers out the corresponding sovaBTC to the user's address. An event is emitted for the withdrawal. Internally, the vault's recorded sovaBTC balance decreases by the redeemed amount.
- **Underlying Collateral Update:** In parallel, the vault's underlying collateral records can be updated. Since the user received sovaBTC, the vault now effectively holds an excess of other BTC tokens (e.g., the WBTC originally deposited). The admin may subsequently remove or rebalance these underlying assets (e.g., swapping WBTC for actual BTC or other wrappers) to manage the reserve. However, those operations are off-chain or manager-executed; from the user's perspective, their obligation is fully met by the sovaBTC payout.

Notably, users **always receive sovaBTC** on withdrawal – they do not redeem directly for the original asset deposited. This unified redemption simplifies the user experience and aligns with Sova's design of sovaBTC as the universal Bitcoin token [11]. It also decouples the vault's collateral mix from redemption flow: regardless of which asset a user contributed, they exit with the standardized sovaBTC. This model allows frictionless conversion between different BTC wrappers via the vault (deposit one type, withdraw another) [3], and facilitates **native BTC yield** strategies by consolidating value in one token [12].

Error Cases: If a user attempts to deposit an unsupported token or a zero amount, the transaction is rejected. If a user tries to redeem more shares than they hold or more value than the vault's available sovaBTC, the transaction reverts with an error (e.g., "insufficient liquidity"). Partial withdrawals are allowed up to available liquidity. Since the vault is permissionless, any Ethereum address can interact; there are no KYC checks or allowlists blocking the flows (unlike the previous FountFi vault which used KYC hook checks on deposit/withdraw [5]).

3. Contracts Overview

The implementation consists of several contracts working in unison:

- **Vault Strategy Contract (`BtcVaultStrategy`):** This is the core contract that holds the underlying BTC tokens and coordinates deposits/redemptions. It inherits from a base Strategy (similar to `ReportedStrategy` in FountFi [13]) but is extended to support multiple asset types. The strategy contract is responsible for maintaining the vault's asset balances, calculating NAV, and enforcing rules on who can withdraw underlying collateral. It also holds references to important sub-contracts like the share token and price reporter.
- **ERC-4626 Share Token Contract (`VaultShareToken`):** An ERC-4626 compliant token (akin to FountFi's `tRWA` token [14]) that represents shares of the vault. This contract mints and burns shares on deposits and withdrawals, respectively. It implements the standard ERC-4626 interface for compatibility (so functions like `deposit`, `mint`, `withdraw`, `redeem` are available), but internally it routes logic to the vault strategy and conversion library to handle multiple asset types. The share token uses 18 decimals to represent fractional shares, even though the underlying assets use 8 decimals, which is handled via scaling math [2]. Each share token is associated one-to-one with a vault strategy instance (the share contract holds an immutable reference to its strategy).
- **Conversion Registry / Library (`ConversionLib` or `AssetRegistry`):** A simple module to manage the list of supported BTC tokens and their conversion rates to the base unit (sovaBTC). In this design, all supported assets share a 1:1 conversion rate with sovaBTC and 8 decimal places, so the registry primarily serves to validate that an asset is allowed and to store its decimals (ensuring they are 8) [4]. The registry could be an on-chain mapping within the strategy (for minimal complexity), or a separate contract if modularity is desired. It provides functions like `isSupportedAsset(address)` and `toBaseUnits(address asset, uint256 amount) -> uint256` which returns the equivalent amount in sovaBTC units (for now, this is typically just the same number, as all are equal and 8 decimal). The registry also can be used to add or remove supported assets by the admin.
- **Price Oracle Reporter (`PriceOracleReporter`):** An off-chain oracle contract that the strategy uses to obtain the current price per share (PPS) for valuation [15]. This contract is maintained by a trusted party or consortium. It allows authorized updaters (e.g. a back-end server or a multi-sig oracle committee) to report a new target price-per-share, and it smooths out changes over time to prevent sudden arbitrage opportunities [16] [17]. The reporter exposes a `report()` function returning the current price per share in 18-decimal fixed-point format [18]. The vault strategy calls this on every valuation (e.g., in `totalAssets()` or when computing shares for deposits) to incorporate any gains or losses of the underlying portfolio. Initially, when the vault is launched, the

price per share will be 1e18 (meaning 1.00000000 sovaBTC per share) [19]. If the vault's collateral is later deployed into yield-generating strategies or revalued, the reporter will update the PPS accordingly, thus affecting how many sovaBTC each share represents.

- **Admin/Manager Account:** While not a contract, it's important to note the role of an admin or manager address in the system. The admin is responsible for critical operations like supplying sovaBTC liquidity for redemptions, initiating certain asset transfers, updating oracle/reporters, and managing supported assets. In the original FountFi, a RoleManager/Registry system defined roles such as STRATEGY_ADMIN and STRATEGY_OPERATOR [20]. In this simplified vault, we reduce complexity by using a straightforward ownership model (e.g., Ownable or a limited role set). The admin (owner) of the strategy will have exclusive access to functions like adding new collateral types, pulling out excess collateral, setting a new PriceOracleReporter (if needed), and pausing the contract in emergencies.

- **Conduit (optional):** The FountFi protocol used a Conduit contract to centralize token transfers for deposits [21] [22]. In our new design, we aim to minimize moving parts. We can either repurpose the Conduit to handle multi-asset deposits (updating its checks to allow any whitelisted token for the calling vault) or integrate transfer logic directly in the vault/share contracts. If the Conduit is reused, users will approve the Conduit for each asset, and the share token will call `Conduit.collectDeposit(token, from, to, amount)` to pull funds from the user [23] [24]. The Conduit ensures the call comes from a valid vault contract and that the token matches an allowed asset for that vault. This approach provides an extra security layer and convenience of a single approval per token. However, for simplicity, an alternative is letting the vault strategy contract call `ERC20.transferFrom` directly (after user approval) for each deposit, removing the need for the Conduit contract entirely. The chosen approach will be guided by security audits and gas optimization considerations.

All contracts will be written in Solidity ^0.8.x, leveraging modern libraries (OpenZeppelin or Solady) for safe math and ERC-20 operations. The vault strategy and share token will closely follow the patterns of the original FountFi contracts (e.g., use of **ERC4626 base classes** [14] and the initialize pattern for upgradability), but unnecessary components (two-phase gated deposits, KYC hooks, complex role hierarchy, etc.) are omitted for clarity and efficiency.

4. Roles and Access Control

This vault employs a **simplified role model** centered on an admin/manager, in contrast to the previous multi-role architecture [20]:

- **Vault Admin (Owner/Manager):** A single address (or multi-sig contract address) is designated as the vault's admin. This role is analogous to the "manager" in the original design [25], but also carries the responsibilities of protocol governance for this particular vault. The admin has the exclusive authority to perform the following actions:
- *Manage Collateral Types:* Add or remove supported BTC tokens in the conversion registry. For example, if a new BTC wrapper (with 8 decimals, 1:1 peg) is introduced, the admin can whitelist its address so users can deposit it. Removing a token would typically be done only if that asset is deprecated or no longer considered equivalent.

- *Liquidity Provision:* Deposit or inject sovaBTC into the vault's strategy as needed to fulfill redemptions. This may involve actually minting or transferring sovaBTC from the reserve system to the strategy contract. The admin can also withdraw surplus underlying assets (WBTC, tBTC, etc.) from the vault to rebalance reserves after users redeem sovaBTC. For instance, if a large amount of WBTC has accumulated in the vault, the admin might remove some and custody it elsewhere once the equivalent sovaBTC has been issued.
- *Oracle Updates:* Set or change the PriceOracleReporter contract address used by the vault (if an upgrade or different reporting mechanism is needed). Only the admin can call the strategy's `setReporter()` function [26]. The admin does **not** directly control the price value, which is managed by the reporter's authorized updaters; however, the admin can switch to a new reporter in case of issues or upgrades (ensuring the reporter is not a single point of failure).
- *Emergency Controls:* Pause or stop the vault in case of an emergency (if a pause mechanism is included). This could prevent new deposits or redemptions temporarily. The admin could also trigger an emergency shutdown where all holdings are returned to the admin for safe unwinding, though this is an extreme measure.

- *Upgrade Management:* If the contracts are upgradeable (see Section 7), the admin has the rights to upgrade the implementation logic or initiate a migration to a new vault.

- **Users:** Regular users (any Ethereum address) have permissionless access to deposit supported tokens and redeem shares for sovaBTC. They do not need any special role or KYC approval – the vault has no KYC hook in place, unlike the original FountFi vaults which integrated compliance hooks to restrict operations [5]. All functions that move user funds (deposit, withdraw, mint, redeem) are open to the public, subject to the vault's standard validation (supported asset, sufficient liquidity, etc.). Users cannot directly withdraw the underlying collateral tokens (WBTC, tBTC, etc.) from the vault – they can only get sovaBTC, by design.

- **Reporter Updaters:** The PriceOracleReporter contract has its own access control for who can post price updates [27] [28]. Typically, the reporter's owner (which could be the vault admin or a governance multi-sig) will add one or more **authorized updaters** (e.g., an off-chain oracle service or a set of signers). These updaters call `update(newPrice, source)` on the reporter to adjust the price per share [29]. This arrangement is external to the vault's contracts but critical to its secure operation. We assume the admin sets this up appropriately (e.g., using a decentralized oracle network or a secure backend to frequently report NAV).

- **No Separate Protocol Roles:** We deliberately avoid introducing separate protocol-level roles like `STRATEGY_OPERATOR`, `PROTOCOL_ADMIN`, or `RULES_ADMIN` that existed in the earlier system [20]. All necessary privileges are consolidated under the vault admin for simplicity. If fine-grained control is needed (for example, if the vault is managed by a DAO), the admin address could be a multi-sig or a smart contract that delegates specific functions to different keys. But from the vault's perspective, it checks only for "admin" on privileged functions.

Access Control Implementation: We will use the OpenZeppelin **Ownable** pattern or a similar role contract. The `BtcVaultStrategy` and possibly the share token will have an `owner` (set at construction or initialization) that corresponds to the admin. Functions like `setReporter`, `addSupportedAsset`, `removeSupportedAsset`, `rescueFunds`, etc., will be protected by an `onlyOwner` modifier. Additionally, certain critical functions on the share token, like adding hooks (if any) or calling certain

ERC4626 overrides, will be restricted to the strategy (since the share token should only accept instructions from its paired strategy contract) [22] [30]. For example, only the strategy is allowed to instruct the share token to transfer assets out via `_collect()` on withdrawal.

Finally, it's worth noting that **redemption operations require coordination with the admin**. In the current design, a user calling redeem will succeed only if the admin has pre-loaded enough sovaBTC into the vault. If the vault is under-collateralized in sovaBTC, users might have to wait for the admin to top it up. In an alternative configuration, we could require the admin (manager) to explicitly initiate withdrawals, similar to the original **ManagedWithdraw** pattern where the manager processes user-signed withdrawal requests [31]. However, to keep the system permissionless and simple, we opt for allowing users to call redeem directly, while clearly documenting that liquidity must be present. The admin is expected to monitor vault liabilities and maintain a buffer of sovaBTC in the strategy to enable smooth withdrawals.

5. Token Handling and Decimals

Consistent handling of token decimals and conversion factors is crucial for a multi-collateral vault. This implementation standardizes all **supported BTC tokens and sovaBTC to 8 decimal places**, mirroring Bitcoin's precision. The vault share token uses 18 decimals, which is conventional for ERC-4626 vault shares and many ERC-20 tokens, to accommodate fractional shares with high precision [9].

Decimals Alignment:

- **Underlying Assets (BTC variants):** Each supported token (WBTC, tBTC, cbBTC, etc.) is expected to have 8 decimals. The vault's conversion registry will reject any asset that doesn't use 8 decimals or cannot be treated as 1:1 equivalent to BTC. This simplifies math since 1.00000000 of any allowed token = 1.00000000 sovaBTC exactly, without additional scaling factors for value. All assets are effectively fungible in terms of value and precision.

- **Base Unit (sovaBTC):** sovaBTC itself has 8 decimals (1 sovaBTC = 1.00000000). The vault internally measures all collateral and NAV in these base units. For example, if the vault holds 100 WBTC and 50 tBTC, it treats that as **150.00000000 sovaBTC** of total assets (assuming those are the only holdings). All NAV reports and calculations use this common unit.

- **Share Token (ERC-4626 shares):** The share token's **decimals = 18**, independent of the asset decimals. This is implemented by overriding the `decimals()` function to return 18, while the underlying asset decimals are 8. In FountFi's original `tRWA` implementation, the relationship between asset and share decimals was handled by an internal offset so that conversions remain precise [32] [33]. We will use a similar approach: for calculations, define `decimalsOffset = 18 (share) - 8 (asset) = 10` and use scaling factors of 10^10 or 10^(18 + 18 - 8) = 10^28 in formulas to convert between share units and asset units [10].

Deposit Calculation: When a user deposits an asset, the vault converts the amount into 18-decimal share units before dividing by price-per-share. For an asset amount `A` (8 dec) and price-per-share `P` (18 dec), shares minted = `A * 10^10 * 1e18 / P / 1e18` = `A * 10^10 / (P / 1e18)`. In practice, the code will do: `shares = (A * 10^10).mulDiv(1e18, P)` to avoid precision loss [10]. For example: - If A = 1.00000000 WBTC (which is 100,000,000 in smallest units) and P = 1e18 (1.0), then shares = 100,000,000 *

10^10 / 1e18 = 100000000 * 10000000000 / 1000000000000000000 = 1e18 shares. - If P = 1.5e18 (1.5 sovaBTC per share due to growth in NAV), depositing 1.00000000 WBTC yields shares = 100000000 * 10^10 / 1.5e18 ≈ 0.6667e18 shares (2/3 of a share), reflecting that each share now represents more underlying. This matches the behavior verified in FountFi's tests for assets with 8 decimals [2] and dynamic prices.

Withdrawal Calculation: Conversely, when shares are redeemed, the vault converts shares (18 dec) back to asset value (8 dec). Using the formula from `ReportedStrategy.balance()` [10], assets = `(pricePerShare * totalShares) / 10^(18 + shareDecimals - assetDecimals)`. For an individual redemption of `S` shares, the amount of sovaBTC returned = `S * P / 10^10` (since shareDecimals=18, assetDecimals=8, so divide by 10^10). If S = 1e18 and P = 1.1e18, assets = 1e18 * 1.1e18 / 1e10 = 1.1e8 (which is 1.10000000 in 8 decimals), as expected.

We ensure that all arithmetic is done with high precision and use safe math libraries (Solady's FixedPointMathLib or OpenZeppelin's Math for mulDiv) to avoid overflow or precision issues [34] [35]. Since the maximum values involved (total BTC in vault, etc.) are far below 2^256, standard 256-bit math with proper scaling is sufficient.

Total Asset Tracking: The vault's total assets (in terms of sovaBTC) is reported via the share token's `totalAssets()` method. This calls the strategy's `balance()` function, which in our design will: 1. Query the reporter for current price-per-share `P` (18 dec). 2. Get the total supply of shares (18 dec). 3. Compute `totalAssets = P * totalSupply / 10^(18 + 18 - 8)` [10]. Because assetDecimals is 8 and shareDecimals 18, this effectively yields a value in 8-decimal units. This approach, inherited from FountFi's design, ensures the vault's view of total assets accounts for any change in NAV. For example, if 100 shares are outstanding and P = 1.05e18, totalAssets = 1.05e18 * 100e18 / 1e28 = 105e8 (i.e., 105.00000000 sovaBTC worth of assets) [36] [37].

Rounding: We will follow ERC-4626's specification for rounding direction on conversions (ERC-4626 mandates certain rounds like round down on withdraw to not take more assets than available, etc.). Given all assets have homogeneous decimals and 1:1 value, rounding issues are minimal, but we'll still comply with the standard: - `previewDeposit(amount)` and `deposit` will round down when calculating shares (so the user might get slightly fewer shares if the division is not exact, leaving a few base units in vault as dust). - `previewRedeem(shares)` and `redeem` will round down the asset amount (ensuring the vault doesn't give out more sovaBTC than the shares' entitlement). The potential dust amounts are on the order of 1 satoshi of BTC or less, which is negligible and can be swept by the admin if needed.

In summary, the vault treats all supported tokens uniformly by standardizing on 8 decimals and 1:1 value. This greatly simplifies the conversion logic and eliminates cross-collateral exchange rates or oracles. The only oracle influence is on overall NAV via the price-per-share, not on the relative value of different BTC tokens (we assume those remain equal by design of the SovaBTC system). This homogeneous handling is a key assumption of the design; if in future a supported asset deviates (say a wrapper depegs), the admin should remove it from allowed assets to protect the vault.

6. NAV and Valuation

Net Asset Value (NAV) in this vault is defined as the total sovaBTC-denominated value of all collateral held, which should equal the total sovaBTC that could be redeemed by all outstanding shares (when the system is liquid). NAV is tracked and updated via a **PriceOracleReporter** mechanism, similarly to the original FountFi reported strategy [5]. However, since all assets are pegged 1:1 to BTC and do not natively fluctuate against each other, changes in NAV will typically come from two sources:

1. **Yield or Gains on Collateral:** If the underlying BTC collateral (WBTC, etc.) is deployed into a yield strategy (off-chain lending, on-chain investment, etc.), the vault's holdings may increase over time (e.g., earning more BTC). The reporter will reflect this by increasing the price-per-share. For example, if the vault starts with P = 1.0 and through yield it gains 5% more BTC, the admin will update the PriceOracleReporter so that it gradually moves to P ≈ 1.05 (meaning each share now entitles the holder to 1.05 BTC units on redemption) [38]. The reporter can smooth this change to avoid instant arbitrage [39], but eventually it will reach the true NAV.
2. **Management Actions and Fees:** If the admin charges a fee or if some collateral is lost (in a rare case of default or hack in underlying strategy), the NAV per share could decrease. The admin would then update the reporter with a lower price-per-share accordingly. Similarly, if new collateral is added that dilutes existing shares (outside of normal deposits), that should be reflected.

PriceOracleReporter Operation: The PriceOracleReporter contract holds the authoritative price-per-share (PPS) value used by the vault. It starts at an initial value (e.g., 1e18) and allows updates by authorized accounts [27]. When an update is submitted via `update(newPPS, source)`, the reporter either sets a new target and transitions to it over a time period (if the change exceeds a configured deviation threshold) [40] [41], or updates immediately if within bounds. This mechanism prevents sudden large NAV shifts that could be exploited. For example, it might limit changes to 1% per hour (configurable by `maxDeviationPerTimePeriod` in basis points) [17]. If a greater change is needed (say NAV jumps by 5%), it will phase it in over multiple periods unless an emergency override (`forceCompleteTransition`) is used [39].

The vault strategy pulls the current price from the reporter whenever it needs to compute balances: - The strategy's `balance()` view function decodes the reporter's `report()` bytes into a uint256 price and performs the calculation of total assets [34]. - The share token's `totalAssets()` uses strategy.balance() [42], thereby indirectly using the latest price. - When a user deposits or withdraws, the share calculations also call the reporter to get `pricePerShare` at that moment (ensuring up-to-date pricing for fairness).

The system assumes the reporter is honest and accurate, which is a centralized trust assumption (mitigated by decentralizing the updater role or using reputable oracles). The admin can swap out the reporter if needed (e.g., replace a slow-moving oracle with a faster updating one or vice versa). Only the vault's manager/admin can call `setReporter()` on the strategy [26] to change the source of NAV data, which emits an event for transparency.

Vault Value in sovaBTC: Because all collateral is valued in sovaBTC terms, the vault's NAV essentially measures how many sovaBTC the vault *should* have for the given collateral. In a fully backed scenario without yield, NAV (in sovaBTC) equals the sum of all actual tokens held (in BTC units). Initially, if no yield or fees, `pricePerShare = 1e18`, so NAV = total shares / 10^(18-8) = total deposited BTC. Over time, if pricePerShare diverges from 1, that indicates NAV changed relative to deposited assets: - If `pricePerShare > 1e18`, the vault has more value than baseline (perhaps due to earned yield or some external BTC injection by admin). - If `pricePerShare < 1e18`, the vault lost value (e.g., a fee was taken or a loss occurred).

In either case, the reporter's updates ensure share holders are affected accordingly. A user's share balance stays constant unless they deposit or withdraw, but the **value** of those shares in sovaBTC changes with NAV. This design where share value floats and is reported externally is directly lifted from the FountFi model of a reported strategy [5], enabling compatibility with various asset management strategies.

One difference in our vault is that **redemptions rely on available liquidity**. The reporter might say each share is worth 1.1 sovaBTC, but if the admin hasn't actually provided that extra 0.1 sovaBTC per share into the vault, users cannot immediately redeem the full value. In an ideal steady-state, the admin will ensure the vault is **always** collateralized to match the reported NAV. This could involve periodically converting the vault's accumulated WBTC/tBTC interest into sovaBTC and depositing it into the strategy. The design assumes a **close coupling between on-chain reported NAV and off-chain reserve management**. Discrepancies should be temporary and resolved by the admin via reserve operations.

Transparency: Every update to the price-per-share is logged (the PriceOracleReporter emits a `PricePerShareUpdated` event with the new target and source [39]), and the vault's total asset value can be queried at any time through `totalAssets()` and `pricePerShare()`. Users can thus monitor the vault's health: the mix of collateral (via on-chain balances of each token in the strategy contract), the outstanding sovaBTC liabilities (should equal totalAssets if fully backed), and the price history of shares.

In summary, NAV is maintained off-chain but enforced on-chain through the reporter's input to the vault calculations. This hybrid approach yields accurate valuations while keeping the vault logic simple (no complex on-chain price calculations). The architecture trusts the reporter and the admin to act in good faith to maintain a 1:1 backing, consistent with Sova's philosophy of transparent, verifiable reserves [43].

7. Upgradeability Considerations

For this vault implementation, we consider two upgradeability approaches:

a. Proxy Upgradeability: Deploy the vault strategy and share token behind proxy contracts (e.g., using UUPS or Transparent Proxy pattern). In this model, the initial deployment uses base implementations, and the admin can upgrade the logic while preserving state (user balances, asset holdings, etc.). Upgradeable proxies provide flexibility to fix bugs or add features post-deployment, important in a complex system. If this route is chosen: - The `BtcVaultStrategy` and `VaultShareToken` contracts will be made **UUPS upgradeable** (or utilize OpenZeppelin's TransparentUpgradeableProxy). The `initialize` function will replace the constructor for setting up initial state (token names, addresses, admin, etc.) [44]. - We must be careful to maintain storage layout compatibility in any upgrades. For instance, if we add a new supported asset in the registry (mapping), that doesn't shift existing storage, but adding new state variables would need to go to the end of the contract's storage. - An access control on upgrades will be enforced: only the vault admin (owner) or a designated governance contract can trigger an implementation upgrade. This ensures no unauthorized party can change vault logic.

b. Fixed Implementation (Non-upgradeable): Alternatively, we might deploy the vault as immutable contracts, as was effectively done in the original FountFi (each strategy/token pair was deployed and not meant to be upgraded in place [45]). In this case, upgrading means deploying a new vault and possibly migrating funds. The benefit is simplicity and eliminating proxy complexity; the drawback is that any change requires user migration or administrator intervention to move assets, which can be disruptive.

Considering the vault's role as a long-term infrastructure (and that it deals with custody of BTC assets), we lean towards a **minimally upgradeable** design. Concretely, we can make only the strategy contract upgradeable, while the share token could remain simple (since its logic is minimal and mostly standard ERC-4626). If a critical bug appears in share token accounting, that would likely require a migration anyway. The strategy, being more complex, is the candidate for proxy upgrade: - The share token would reference the strategy via an immutable pointer. If the strategy is upgraded, the share token continues to point to the proxy (so it now uses the new strategy logic seamlessly). - We must ensure that an upgrade does not invalidate assumptions between the share and strategy. We will write extensive tests to confirm that after an upgrade, deposits/withdrawals still work and no user balances are lost or mis-calculated.

We will also implement **upgrade guards**: for example, using OpenZeppelin's UUPS `_authorizeUpgrade` to restrict who can upgrade, and possibly require a time-lock or multi-sig confirmation outside the contract before calling upgrade, to add security.

State Migration: If for some reason a proxy upgrade is not feasible (or is deemed too risky), the fallback path is a manual migration. The admin could: 1. Deploy a new vault contract (with the fixed code). 2. Pause the old vault, preventing new deposits. 3. Use an admin function to transfer all underlying assets from the old vault to the new vault (this requires a function to allow admin to withdraw underlying tokens, which we plan to include). 4. Initiate a process to swap users' old shares for new shares (perhaps by letting them redeem old shares for sovaBTC, then deposit sovaBTC into the new vault for new shares at equivalent value). This process is cumbersome, so we consider it only as a last resort. It's mentioned here to illustrate that even if upgradability is in place, a catastrophic bug might call for such a migration if upgradability fails.

Given the focus on "reuse as much as possible" from the original implementation, we note that FountFi's `Registry.deploy()` method was used to create new strategy instances from templates [45]. Our design might not need a central factory if only one vault is being launched. But if multiple such vaults are planned (e.g., separate vaults for different asset groups), we could repurpose a simplified factory or the existing Registry. In that case, deploying a new vault type would involve adding its implementation to the factory's allowed list and calling deploy. However, since our scope is one multi-collateral BTC vault, we can directly deploy the pair and set the admin.

In conclusion, the vault will either be upgradable via proxy or at least easily redeployed. We will document the chosen approach clearly and ensure the community or stakeholders understand the trust model (if upgradable, users must trust the admin not to maliciously upgrade to faulty logic; if not, users must be aware that migration might be needed for upgrades). Considering security, an audit will specifically review the upgradeability mechanisms to prevent vulnerabilities.

8. Design Constraints (Simplicity and Testing Compatibility)

This implementation is explicitly designed to be **simple, modular, and testable**, avoiding the complexity pitfalls of the prior iteration. Key design constraints and choices include:

- **Minimize Contract Size and Complexity:** We aim to reduce function size and nesting levels to avoid the Solidity "stack too deep" errors that occur when too many variables are in scope [6]. In the original code, workarounds like caching struct arrays were used to mitigate stack depth issues [7]; our goal is to structure the code so that such workarounds are largely unnecessary. Each function will have a clear, singular purpose. For example, the deposit flow might be broken into sub-functions: `_transferIn(asset, amount)`, `_mintShares(to, shares)`, etc., each with limited local variables, rather than one monolithic function doing all steps. We will utilize helper libraries for math (e.g., a library to handle the mulDiv operations for share<->asset conversions) and for validation (e.g., a library function to verify an asset is supported and retrieve its decimals). These helpers encapsulate logic and keep the main contract functions clean and under the local variable limit. This modular approach improves readability and maintainability, while making the code less prone to compiler limitations [46] .

- **Avoid Deep Nesting:** The contract logic is kept as flat as possible. Instead of nested `if/else` or loops inside loops, we use early `require` checks and external calls to libraries. Removing the entire KYC/Hook system eliminates multiple layers of nested checks that previously ran on each operation (like iterating through hooks for deposits and withdrawals) [47] [48] . By operating in a permissionless mode, we drop the need for the `RulesEngine` and `IHook` interfaces that contributed to call depth and stack usage. The result is a straightforward flow: check inputs, update state, transfer tokens. This not only helps with stack depth but also ensures **Forge coverage** runs smoothly. In the past, instrumentation for coverage made some functions hit stack too deep errors during testing [49] ; by simplifying functions, we mitigate that risk. Our testing plan (below) will include running coverage to ensure no such errors occur.

- **Use of Libraries and Inheritance:** We leverage battle-tested libraries from Solady/OpenZeppelin for low-level operations (SafeTransfer, math) so we don't inline too much code. The share token inherits from Solady's ERC4626 which already provides base implementations for many functions, but we override where necessary to implement multi-asset support. We will be careful in overriding to not introduce reentrancy issues – reentrancy guards are used on state-changing functions as in the original (nonReentrant modifiers are present on deposit/withdraw internals) [47] [50] . Reentrancy guard and other utility storage will be placed in the strategy contract to keep the share token minimal.

- **Gas and Efficiency:** Simplicity also aids gas efficiency. By removing needless indirection (e.g., calling through a Registry for every deposit) and conditionals (no per-hook validations), each deposit or redeem call touches only the necessary storage and logic. The conversion registry lookup is a single mapping access. The pricing call to reporter is an external call, but that's essential for correctness. We will cache the reporter value within a transaction to avoid multiple calls if needed (although in our model typically only one reporter call is needed per user operation, e.g., during balance calculation).

- **Full Test Coverage:** The design will be accompanied by a comprehensive test suite (see next section). We will pay special attention to edge cases (zero amounts, extreme large deposits, price per share with many decimal places, etc.). For instance, we will test a scenario where price-per-share is a very precise value (the original tests included scenarios like 1.123456789012345678 to ensure no precision loss). We will also test multiple sequential operations to ensure state integrity.

- **Solidity Version and Features:** Using Solidity 0.8.18+ (targeting 0.8.25 as in original [51]) gives us safe math by default and other enhancements. We will use custom errors (as seen in original code for efficiency) for revert reasons like `UnsupportedAsset()` or `InsufficientLiquidity()`, saving gas over revert strings. We avoid using deprecated features. Where possible, constants and immutables will be used to save gas (e.g., the share token's decimals can be constant 18; asset decimals constant 8).

- **Static Analysis and Linting:** The code will be kept simple also to make it easier to analyze. We will run static analysis (Slither, etc.) to catch any potential issues early. Simpler code reduces the chance of something subtle slipping through.

In summary, the contract design philosophy is **"make it as simple as possible, but no simpler."** We provide all required functionality for a multi-collateral vault, but strip out any extraneous mechanism from the legacy design. This yields a cleaner implementation that is easier to audit and less error-prone. The improvements directly address prior pain points such as stack-too-deep compiler errors and overly complex control flow, resulting in a robust yet elegant vault contract that is compatible with development tools (coverage, fuzzing) and can be confidently maintained.

9. Test and Deployment Plan

A thorough testing and deployment plan will ensure the new vault is reliable and secure:

Unit Testing: We will write extensive unit tests covering all core scenarios:

- *Deposit and Redemption:* Test single-asset deposit (e.g., user deposits WBTC, verify their share balance increases appropriately, vault's internal asset mapping updated, events emitted). Then test redemption by the same user after providing sovaBTC liquidity: ensure they receive exactly the expected sovaBTC and their share balance goes to zero. Do this for each supported asset variant to ensure no asset-specific bugs.
- *Multi-Asset Pooling:* Test deposits of different asset types by different users without intermediate withdrawals. For example, Alice deposits 1 WBTC, Bob deposits 1 tBTC, Charlie deposits 1 cbBTC. Verify that vault totalAssets = 3.0 sovaBTC (in 8 decimals) and total shares issued correspond to 3.0 value (depending on pricePerShare). Then test partial withdrawals: e.g., Alice withdraws half her shares and gets 0.5 sovaBTC. Verify vault still holds 1 WBTC (Alice's remaining) + others, and now also has 0.5 sovaBTC less in liquidity.
- *Price Per Share Effects:* Simulate a NAV increase by manipulating the reporter. For instance, deposit assets, then have the PriceOracleReporter's authorized updater set a new target price, e.g., +10%. Call a function (or advance time and call multiple updates if gradual) to update the price. Now test that `totalAssets()` reflects the increased value (should be 1.1x prior) and that a new deposit after the price change yields fewer shares (which we can assert approximately or exactly, as in the original tests for price change scenarios [52] [37]). Also test that an existing user redeeming after the price increase gets proportionally more sovaBTC than if they had redeemed before (testing that the vault honors the current price for all).
- *Decimals and Rounding:* Test edge cases around decimal conversion. For example, deposit 1 satoshi (0.00000001) of WBTC – the smallest unit – and ensure it mints some shares (it will likely mint a very small fraction of a share). Then redeem that share fraction and verify you get back 1 satoshi of sovaBTC (or it may round down to 0 if too tiny – we need to define expected behavior for dust). Test a scenario with a very high total supply of shares and ensure no overflow in calculations (we can simulate by minting shares directly to an address for test purposes to push the limits).
- *Unsupported Assets:* Attempt to deposit a token not in the conversion registry (e.g., an ERC20 with 8 decimals that isn't whitelisted, or an ERC20 with 18 decimals) and expect a revert with

`UnsupportedAsset`. Also test removal of an asset: add a dummy asset in registry, deposit it, then admin marks it unsupported and try depositing again (should fail). Verify that existing deposits of that asset are still counted in NAV (they are still in vault) but users cannot add new ones.

- *Admin Functions:* Test admin-only operations:
- Add a new asset type: call `addSupportedAsset` as admin, deposit that asset as user, ensure it works.
- Remove asset type: as above, ensure only admin can call it, and after removal deposits are blocked.
- Provide liquidity: simulate admin transferring sovaBTC to the strategy (this could be done by directly calling the token contract's transfer, since strategy is a normal address). Then a user redeems shares – verify the strategy's sovaBTC balance decreases accordingly.
- Emergency withdrawal (if implemented): e.g., admin calls `withdrawCollateral(token, amount, to)` to pull out underlying WBTC from the vault. Ensure it reduces vault holdings and logs an event. Only admin should succeed; a user calling it should revert.
- Change reporter: set a new reporter contract address, and ensure the strategy now uses the new one for price. Perhaps test by having a mock reporter that always returns a fixed price to see the switch effect.
- *Reentrancy and Security:* Use Foundry's fuzz testing to attempt reentrancy if any (likely none, but for example, ensure that if a malicious token had a callback on transfer, our nonReentrant prevents reentrant calls on deposit/withdraw). Also test that a user cannot withdraw more than their share (maxRedeem logic in ERC4626 should cover it, but we double-check).
- *Multiple Users and Fuzz:* Write a multi-user scenario where a sequence of random deposits and withdrawals are done (fuzz test with constraints) and then at the end check invariant: total sovaBTC paid out + current vault holdings should equal total sovaBTC ever deposited (minus any yield accounted by reporter). This ensures conservation of value.
- *Coverage:* Run `forge coverage` to ensure we hit all code paths, including failure cases. The simplified code should now compile under coverage without issue; if any function trips stack-too-deep in coverage, we will refactor it.

Integration Testing: After unit tests, we simulate an end-to-end integration on a local fork or test network: - Deploy the PriceOracleReporter, set an updater. - Deploy the sovaBTC token contract (if not already deployed; possibly we treat sovaBTC as an existing token on the network). - Deploy the vault (strategy and share token). The admin will initially add assets like WBTC, tBTC with their real addresses (on testnet or mainnet fork). - Mint some WBTC/tBTC to test user addresses (or use testnet faucets). - Have test users deposit and redeem through the full UI flow (if available) or via simulated transactions. This tests the interaction with real token contracts (ensuring things like `safeTransferFrom` works with actual tokens, and checking that token approvals and decimals are handled correctly). - Test the actual minting of sovaBTC: in a real scenario, perhaps the admin deposit of sovaBTC means calling the actual sovaBTC contract's mint function (if the admin is authorized to mint). On testnet, we may simulate this by just transferring from a preminted supply. Ensure that when users redeem and get sovaBTC, those tokens are indeed transferable and match expected balances.

Audit and Formal Verification: We will undergo a professional audit of the contracts, focusing on the multi-collateral logic, access control, and economic consistency. Any findings will be addressed and tests added for those cases. If feasible, we'll also set up formal verification for critical properties (e.g., no loss of funds invariants, only admin can perform admin actions, etc.).

Deployment Plan: 1. **Testnet Deployment:** Deploy the vault on a test network (e.g., Base Goerli or Ethereum Sepolia if appropriate). Use real token addresses for WBTC (testnet version) and others, or deploy mock tokens that simulate them (8 decimals). Conduct a public test with a small amount of funds to observe behavior. This will involve the intended admin/multi-sig controlling the vault and a few users. 2. **Parameter Tuning:** Based on testnet results, adjust any parameters (like reporter deviation settings, any fee switch if applicable which currently is not in spec, etc.). Ensure the conversion registry is correctly initialized with all intended collateral types before mainnet. 3. **Mainnet Deployment:** Deploy the PriceOracleReporter (or use an existing one if shared), then deploy the vault strategy and share token. Initialize with: - Name/Symbol for share token (e.g., "Sova BTC Vault Share", "sovaVaultBTC" or similar). - The asset (base asset) could be set to sovaBTC address, and assetDecimals=8. - Set the admin address (multi-sig controlling the vault). - Set the PriceOracleReporter address via init data or a subsequent tx. - Register supported assets: call `addSupportedAsset(WBTC_address, 8)`, etc., for each variant. - Ensure the Conduit (if used) knows about the new share token (in case a central Conduit is used, it might automatically allow any `isStrategyToken`). - Mint or deposit an initial amount of sovaBTC liquidity to the vault (so early users can redeem immediately). For instance, the admin might pre-fund, say, 10 sovaBTC to the strategy contract as a starting buffer. 4. **Security Checks:** After deployment, test a small deposit and redemption on mainnet (with admin participation) to verify everything works with real tokens and that events/logs look correct. Monitor that the PriceOracleReporter updates are coming through (the reporter might initially just report 1.0 until something changes). 5. **Monitoring:** Set up monitoring for the vault contract – track total assets, price per share, and liquidity levels. If the vault approaches low sovaBTC liquidity (e.g., lots of shares outstanding vs sovaBTC available), alerts should prompt the admin to refill liquidity or suspend new deposits if something's wrong. 6. **Documentation and User Guide:** Publish documentation explaining how users can interact (this largely follows standard ERC-4626 usage for deposits/withdrawals, but noting the need to approve the Conduit or vault, and that redemptions yield sovaBTC). Also outline the transparency mechanisms: where to see proof of reserves (on-chain holdings of the vault, which are public), how the reporter updates can be tracked via events, etc.

By following this test and deployment plan, we ensure that the multi-collateral BTC vault is launched with confidence. The comprehensive tests give a high assurance of correctness, and the gradual rollout (testnet to mainnet with careful monitoring) minimizes risk. This vault will provide a foundational piece of the Sova/FountFi ecosystem: a secure, flexible bridge between various BTC wrappers and the unified sovaBTC token, enabling users to seamlessly move BTC value on-chain [3] and participate in BTC-native DeFi yields [12] with simplicity and security.

Sources:

1. FountFi Protocol Technical Documentation [1] [5] [20]
2. FountFi Contracts (tRWA, ReportedStrategy, Conduit, ManagedWithdraw, etc.) [14] [10] [8] [31] [7]
3. Sova Network Documentation on sovaBTC [4] [3]
4. FountFi Test Cases (asset decimals handling, price per share updates) [2] [38]
5. Stack Exchange on Solidity "stack too deep" error [6] and Foundry coverage issue [49]

---

[1] [5] [13] [20] [45] README.md
https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/README.md

2 9 19 25 36 37 38 52 ReportedStrategy.t.sol
https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/test/ReportedStrategy.t.sol

3 4 11 12 43 SovaBTC | Sova Docs
https://docs.sova.io/sovabtc

6 Need Help Debugging "Stack too deep" Error in Solidity with Forge ...
https://github.com/Cyfrin/foundry-full-course-cu/discussions/851

7 ManagedWithdrawRWA.sol
https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/src/token/ManagedWithdrawRWA.sol

8 14 32 33 42 47 48 50 51 tRWA.sol
https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/src/token/tRWA.sol

10 26 34 35 44 ReportedStrategy.sol
https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/src/strategy/ReportedStrategy.sol

15 16 17 18 27 28 29 39 40 41 PriceOracleReporter.sol
https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/src/reporter/PriceOracleReporter.sol

21 22 23 24 30 Conduit.sol
https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/src/conduit/Conduit.sol

31 ManagedWithdrawRWAStrategy.sol
https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/src/strategy/ManagedWithdrawRWAStrategy.sol

46 Solidity Gotchas — Part 4: Stack too deep | by Simon Palmer | Medium
https://medium.com/@simon.palmer_42769/solidity-gotchas-part-4-stack-too-deep-929a0b488730

49 Stack too deep error with forge coverage , even with --via-ir , optimizer and --ir-minimum flag · foundry-rs foundry · Discussion #8766 · GitHub
https://github.com/foundry-rs/foundry/discussions/8766