ChatGPT

FountFi Multi-BTC Vault Implementation Guide

Architecture Overview

System Goals: Build a vault that accepts **multiple BTC-pegged ERC-20 tokens** (e.g. WBTC, tBTC, etc., all 8 decimals) as deposits and **always redeems in sovaBTC** (an 8-decimal token representing 1 BTC within the Sova ecosystem). The vault issues an **ERC-4626 compliant share token** (receipt token) with **18 decimal places** to depositors. Yields from off-chain or managed strategies accrue via **NAV (Net Asset Value) updates** reported by a centralized oracle feed (the `PriceOracleReporter`). The design heavily leverages existing FountFi components for consistency and upgradeability, including role-based access control and deposit "conduit" mechanics.

Key On-Chain Components:

- **Multi-Collateral BTC Vault (ERC-4626)** – The core vault contract (upgradeable) that implements deposit/withdraw logic. Under the hood it uses **sovaBTC as the unified underlying asset** (8 decimals) but accepts various BTC derivative tokens by converting their value to sovaBTC. It mints/burns **18-decimal ERC-4626 share tokens** (e.g. **mcBTC** for "Multi-collateral BTC") to represent depositor ownership. This contract interfaces with the PriceOracle for share<->asset conversions and with a Strategy contract for asset custody.

- **MultiCollateralRegistry** – A registry contract listing all **approved BTC collateral tokens** and their conversion rates to sovaBTC. For each token, it stores the token's decimals and a conversion factor (scaled to 1e18) representing how many sovaBTC units 1 unit of that token represents. In our case, all BTC derivatives start with a **1:1 rate** (1 token = 1 sovaBTC) [1], but the registry allows adjusting rates if needed (e.g. to discount a collateral's value if it deviates from BTC parity). This module standardizes value conversion across different token decimals.

- **PriceOracleReporter (NAV Oracle)** – A trusted oracle contract providing the **price per share** (NAV) of the vault in real time. Instead of calculating yield on-chain, the vault relies on this oracle's reported price to determine how much underlying each share is worth. The oracle supports **gradual price updates** to prevent arbitrage [2]. Initially, the price per share will be set to **1e18 (with 18 decimal precision)**, meaning 1 share = 1 sovaBTC [3] [4]. Authorized updaters (e.g. admins or off-chain agents) can call `update(newPrice, source)` to adjust the NAV over time as yield accrues off-chain. The vault will query this oracle's `report()` or `getCurrentPrice()` value when handling deposits and redemptions.

- **Strategy Contract (MultiCollateralStrategy)** – The upgradeable strategy logic that actually **holds the deposited assets** (sovaBTC and other BTC tokens) and potentially interfaces with off-chain yield generation. In this design, the strategy acts as a custodian: all user deposits are forwarded to it, and it will supply sovaBTC for redemptions. It is paired 1:1 with the vault (the vault uses a single strategy at a time). The strategy can be minimal (just holding assets) or implement more complex behavior (e.g. periodic conversion of WBTC/tBTC to sovaBTC, or sending assets off-chain). For simplicity, we

start with a basic strategy that holds all collateral tokens and doesn't deploy them elsewhere – yield is handled by oracle NAV updates rather than on-chain investments. The strategy is still useful as a separable module that can be **upgraded** independently (deploy new strategy logic and shift funds) without replacing the vault or its token contract.

- **Conduit (Deposit Collector)** – A **centralized token transfer conduit** that streamlines user deposits. In FountFi, the Conduit allows users to approve a single contract to spend their assets, and then the vault can pull deposits via that conduit [5]. The Conduit contract is set up with the vault's address as an authorized caller (often referred to as an "approved tRWA contract" in FountFi terms). When a user deposits, the vault will call the Conduit's `collectDeposit(token, from, to, amount)` function to transfer tokens from the user (`from`) to the target (`to`, which will be the Strategy or vault) [5]. This design means users only manage one approval (to the Conduit) for all supported tokens, and it centralizes custody transfers for security (with the Conduit performing standardized safety checks on amounts and destinations). The Conduit itself inherits role-based access (via `RoleManaged` / `RoleManager`) so that only approved vaults can call it, and only admins can rescue tokens or manage approvals [6].

- **RoleManager & Access Control** – A global role management contract (`RoleManager`) that defines various roles (most as `bytes32` constants) and keeps track of addresses that have those roles. All core contracts (Vault, Strategy, Conduit, Registry) will be integrated with RoleManager via an abstract `RoleManaged` base, allowing use of modifiers like `onlyRoles(roleManager.PROTOCOL_ADMIN())` to restrict admin functions [6]. Key roles likely include:

  - `PROTOCOL_ADMIN` – superuser for configuring contracts (e.g. add collateral types, perform upgrades, emergency actions).
  - `VAULT_ADMIN` – perhaps a subset for vault-specific admin actions (if separated from protocol-wide admin).
  - `STRATEGY_OPERATOR` – for addresses/contracts allowed to execute strategy operations (in FountFi, the strategy contract itself is granted this role so it can perform certain calls) [7].
  - `REPORTER_UPDATER` – if integrated, could be used to gate who can update the PriceOracleReporter (though PriceOracleReporter also has its own internal authorizedUpdaters list).

  - `APPROVED_VAULT` or similar – a role that the Conduit checks to allow a vault to call `collectDeposit`. (The exact naming depends on the RoleManager setup; in practice, **after deploying the vault we will grant it the role that Conduit expects for authorized callers**.)

- **Upgradeable Proxy Architecture** – Both the Vault and Strategy will use an upgradeable deployment (e.g. EIP-1967 proxy pattern or UUPS). This ensures the core logic can be improved or patched without replacing the user's tokens or requiring migration. In FountFi, the `Conduit or standard proxy` phrasing suggests we can either use their existing upgrade pattern or a standard OpenZeppelin proxy. We will use a **Transparent Proxy** for clarity:

  - Deploy the **Vault implementation** (logic contract) and **Strategy implementation** logic contracts.
  - Deploy proxy contracts pointing to each, controlled by an admin (could be a ProxyAdmin contract governed by RoleManager's PROTOCOL_ADMIN).

- Initialize the proxies (call an `initialize(...)` function on Vault and Strategy) to set up state (addresses for RoleManager, Conduit, registry, oracle, etc.). After init, the vault proxy will represent the **mcBTC vault** and users will interact with this proxy address.
- Use RoleManager roles to restrict the upgrade function (for Transparent proxies this is handled via ProxyAdmin; if UUPS, we'd include `onlyRoles(PROTOCOL_ADMIN)` in the `upgradeTo` function).

Storage Layout & Upgrade Safety: The Vault and Strategy implementations should include storage gap padding (e.g. `uint256[50] private __gap;`) if using UUPS, or simply avoid state in the proxy beyond what's needed. When upgrading, **do not modify the ordering of existing state variables** and only append new variables to preserve layout. All critical state (like user balances in the ERC20 vault) resides in proxy storage, so upgrades must be done carefully to avoid collisions. It's recommended to write tests or use a tool to ensure that the storage layout in new versions is compatible with the old.

Component Relationships:

- **Vault ↔ Oracle:** The vault queries the PriceOracleReporter on every user deposit and withdrawal to convert between shares and underlying assets. The oracle's `currentPricePerShare` (18-decimal fixed-point) is treated as the source of truth for vault NAV. For example, if the oracle reports 1.05 * 1e18, then each share token represents 1.05 sovaBTC. This influences how many shares to mint for a deposit or how many sovaBTC to pay out per share on redemption.

- **Vault ↔ Registry/Strategy:** The vault itself **does not hold any collateral tokens** (to simplify accounting and security); instead, all deposited tokens are immediately sent to the Strategy contract for storage. The vault keeps track of total assets via the Strategy or oracle rather than by holding tokens. The MultiCollateralRegistry is used by the vault (or strategy) to validate and convert incoming deposit amounts. On deposit, the vault uses `registry` to compute the sovaBTC-equivalent of the incoming token and then instructs the Conduit to move the tokens into the strategy. On withdrawal, the vault will instruct the strategy to release sovaBTC to fulfill redemptions (the strategy may internally trade or draw from its reserves of sovaBTC and other tokens to satisfy this).

- **User ↔ Vault (via Conduit):** For a user deposit, the flow is:

- **Approval:** User calls `ERC20.approve(Conduit, amount)` on the token they want to deposit (WBTC, tBTC, etc.). They only need to do this once per token, since Conduit can be reused for all vault deposits.
- **Deposit:** User calls the vault's deposit function (specifying which token and amount). The vault (which is an approved caller in Conduit) will invoke `Conduit.collectDeposit(token, user, strategy, amount)`. The Conduit then transfers `amount` of `token` from `user` to the `strategy` contract's address, assuming the user's approval and balance are sufficient [5]. This **pull pattern** ensures the transfer respects the central approval and routing rules.
- **Mint Shares:** Once the transfer is confirmed, the vault mints the appropriate amount of **mcBTC vault shares** (18-dec ERC20) to the user's account, representing their claim on the underlying.

On **redemption (withdrawal)**, the user calls the vault's withdraw or redeem function. The vault will calculate the sovaBTC owed for the given shares (using the oracle price), burn the shares, and instruct the strategy to send that amount of sovaBTC to the user's address. The user ends up with sovaBTC, regardless of what token they originally deposited. (Operationally, the strategy might internally hold a mix of WBTC/

tBTC and needs to have enough sovaBTC liquid to redeem – in practice, the protocol team would periodically rebalance the strategy's holdings so that redemptions can be honored, possibly by swapping or otherwise obtaining sovaBTC using the deposited collaterals.)

- **Admin ↔ System:** Administrators (addresses with `PROTOCOL_ADMIN` role, likely the deploying entity or a multisig) have broad control:
- They can **add new collateral token types** or adjust conversion rates in the `MultiCollateralRegistry` (e.g. adding a new BTC variant or changing the rate if one token deviates from 1 BTC value).
- They can update parameters on the PriceOracleReporter (like max deviation or transition speed) or push new NAV updates (e.g. updating price per share to reflect earned yield) – only addresses marked as authorized updaters in PriceOracleReporter can do this [8].
- They can **pause** deposits or withdrawals in emergencies if we implement a pause mechanism (this would be guarded by a role like `PAUSER` or admin).
- They can initiate **upgrades** to the Vault or Strategy logic by deploying new implementations and pointing proxies to them (only the ProxyAdmin or upgrade role can do this, to prevent unauthorized changes).
- They can grant or revoke roles via RoleManager. For example, if deploying a new strategy contract, an admin would grant it the `STRATEGY_OPERATOR` role so it can function fully [7], and might revoke that role from an old strategy if it's being retired.
- **No KYC hooks:** Unlike some RWA vaults, this system does *not* enforce any whitelist or KYC verification on user actions. All deposit and redeem functions are permissionless for any user (only governed by the usual token approvals and vault status). We deliberately avoid any off-chain signature or KYC gating in the smart contracts. (In FountFi's RWA context, there were "ManagedWithdraw" strategies requiring signed approval; here we omit such features for simplicity and openness.)

Below is a high-level interface sketch of the main Vault contract and its collaborators, illustrating the relationships and key functions with comments:

```
interface IMultiCollateralRegistry {
    /// @notice Register a new collateral token and its conversion rate to
sovaBTC.
    /// @param token The collateral token address.
    /// @param conversionRate Ray-scaled (1e18) conversion to underlying
(sovaBTC).
    ///                       E.g. 1e18 means 1 token = 1 sovaBTC [1].
    /// @param decimals The decimals of the token (for scaling).
    function addCollateral(address token, uint256 conversionRate, uint8
decimals) external;

    /// @notice Get the equivalent underlying (sovaBTC) amount for a given
collateral token amount.
    /// @param token The collateral token address.
    /// @param amount The amount of the collateral token (in its native smallest
units).
    /// @return underlyingAmount The value in sovaBTC base units (8-decimal
```

```
base).
    function getValueInUnderlying(address token, uint256 amount) external view
returns (uint256 underlyingAmount);

    // ... other methods like removeCollateral, getCollateralInfo, etc.
}

interface IPriceOracleReporter {
    function getCurrentPrice() external view returns (uint256);  // current
price per share (18-decimal)
    function report() external view returns (bytes memory);      // ERC
interface that returns encoded price
    function update(uint256 newPrice, string calldata source) external;  //
authorized callers only
    // ... (PriceOracleReporter is Ownable, and has setUpdater, etc. for admin
control) 8
}

interface IConduit {
    /// @dev Only callable by authorized vaults (RoleManaged via RoleManager).
    function collectDeposit(address token, address from, address to, uint256
amount) external returns (bool);
    function rescueERC20(address token, address to, uint256 amount)
external;  // admin-only to recover stuck funds 6
    // ... (Conduit is initialized with RoleManager address and uses it for auth
checks)
}

interface IRoleManager {
    function grantRole(address target, bytes32 role) external;
    function revokeRole(address target, bytes32 role) external;
    function PROTOCOL_ADMIN() external view returns (bytes32);
    function STRATEGY_OPERATOR() external view returns (bytes32);
    // (Plus any other roles like vault roles, etc.)
}

/// @notice Interface of the upgradeable Strategy. It holds all collateral
tokens.
interface IMultiCollateralStrategy {
    function withdrawTo(address asset, address to, uint256 amount) external;
    // ^ Called by vault to pull out underlying sovaBTC for redemption.
    function setVault(address vault) external;
    // ... (plus perhaps functions to handle adding new collateral types, if
needed)
}
```

And the **Vault (ERC-4626)** itself (simplified interface and critical logic):

```
interface IMultiBTCVault {  // conforms to IERC4626 for core functions
    /// @notice Address of the underlying redemption asset (sovaBTC)
    function asset() external view returns (address);  // sovaBTC address (8
decimals)

    /// @notice Deposit any supported BTC derivative token to receive vault
shares.
    /// @param token The address of the BTC collateral token to deposit (must be
whitelisted in registry).
    /// @param amount The amount of `token` to deposit (in the token's native
decimals).
    /// @param receiver The address to receive minted vault share tokens.
    /// @return shares The number of vault shares (18-decimals) minted to
`receiver`.
    function deposit(address token, uint256 amount, address receiver) external
returns (uint256 shares);

    /// @notice Redeem vault shares for **sovaBTC** (underlying) withdrawal.
    /// @param shares The number of vault share tokens to redeem (18-decimal
units).
    /// @param receiver The address to receive the redeemed sovaBTC.
    /// @param owner The address that owns the shares (if different from
msg.sender, an approval is required).
    /// @return assets The amount of sovaBTC released to `receiver` (in 8-
decimal base units).
    function redeem(uint256 shares, address receiver, address owner) external
returns (uint256 assets);

    /// @notice (ERC-4626) Convert a given amount of sovaBTC (underlying) to
shares, using current price.
    function convertToShares(uint256 assets) external view returns (uint256
shares);

    /// @notice (ERC-4626) Convert a given amount of shares to sovaBTC
(underlying), using current price.
    function convertToAssets(uint256 shares) external view returns (uint256
assets);

    /// @notice Total sovaBTC equivalent managed by this vault (sum of all
collateral, in underlying terms).
    function totalAssets() external view returns (uint256);

    // ... other standard ERC-4626 functions: maxDeposit, maxWithdraw, etc.,
likely all permissionless.
}
```

Vault Internal Logic Highlights: The vault implements the above interface. Key operations are:

- **Initialization:** The vault is initialized with addresses of the RoleManager, PriceOracleReporter, MultiCollateralRegistry, Conduit, Strategy, and the underlying asset (sovaBTC) and its decimals. For example, the constructor or initializer would store `sovaBTC` as the underlying asset and perhaps verify its decimals = 8. The share token metadata (name, symbol, decimals=18) is set here as well (if using OpenZeppelin's ERC20, we override `decimals()` to return 18).

- **Deposit (multi-asset):** When `deposit(token, amount, receiver)` is called:

- **Validate Collateral:** Check that `token` is an allowed collateral in the registry (e.g., `require(registry.getValueInUnderlying(token, 1) > 0, "Unsupported asset")` or a dedicated registry check function).
- **Convert to Underlying Value:** Use the registry to compute how much underlying (sovaBTC) this deposit is worth. For example:

```
uint256 underlyingValue = registry.getValueInUnderlying(token, amount);
```

  Here `underlyingValue` is in **sovaBTC's smallest units (satoshis)**. If `token` has 8 decimals and 1:1 rate, depositing 1e8 of it yields `underlyingValue = 1e8` (one sovaBTC's satoshis). If the token has 18 decimals (like some tBTC versions), the registry will internally account for the decimal difference so that the result is still 1e8 for one whole token, given a 1e18 conversion rate [9].
- **Calculate Shares:** Determine how many vault shares to mint for this deposit. We use the oracle's current price per share:

```
uint256 currentPrice = priceOracle.getCurrentPrice();  // 18-decimal value
// Scale underlying value to 18-decimal for division:
uint256 underlyingValueScaled = underlyingValue * (10**(18 - 8));
// (If sovaBTC has 8 decimals, multiply by 10^(18-8)=10^10 to convert to
18-decimal scale)
uint256 sharesToMint = (underlyingValueScaled * 1e18) / currentPrice;
require(sharesToMint > 0, "Deposit too small for a share");
```

  This formula ensures that if the NAV is above 1, the user gets proportionally fewer shares (since each share is more valuable). For example, if 1 share = 1.10 sovaBTC (currentPrice = 1.10e18), depositing 1.00 sovaBTC worth will mint ~0.909 shares. **This preserves fairness** so existing shareholders are not diluted by new deposits when NAV has grown.
- **Mint Shares:** Call `_mint(receiver, sharesToMint)` to issue new share tokens to the depositor's address. The vault's share token is an ERC20, so this updates balances and total supply.
- **Transfer Collateral In:** Use the Conduit to actually pull the user's tokens:

```
bool success = conduit.collectDeposit(token, msg.sender,
address(strategy), amount);
require(success, "Conduit deposit failed");
```

This will transfer the specified `amount` of `token` from the user (`msg.sender`) into the Strategy contract (`to = address(strategy)`). The vault itself never directly holds the tokens. The `collectDeposit` call will fail if the user didn't approve the Conduit or doesn't have enough balance, or if `vault` is not authorized as a caller (hence we must set that up in advance).

- **Emit Deposit event:** (As per ERC-4626, emit a `Deposit(msg.sender, receiver, tokenAmount, sharesMinted)` event for logging).

- **Withdrawal/Redemption:** We implement **redeem(shares)** and **withdraw(assets)** focusing on sovaBTC output:

- *Redeem (preferred by ERC-4626):* User specifies a number of shares to redeem. The vault will:

  1. Calculate the underlying amount owed: `assets = convertToAssets(shares)`. Using the oracle price,

     ```
     uint256 currentPrice = priceOracle.getCurrentPrice(); // 18-dec
     uint256 underlyingScaled = (shares * currentPrice) / 1e18;
     uint256 assets = underlyingScaled / (10**(18 - 8));
     // Now `assets` is in sovaBTC base units (8 decimals).
     ```

     This gives the amount of sovaBTC (sat) that the shares represent at the current NAV.
  2. Burn the shares from the owner's balance: `_burn(owner, shares)` (handling approval if `owner != msg.sender` as per ERC20 standard).
  3. Instruct the strategy to send out the sovaBTC: e.g. `strategy.withdrawTo(sovaBTC, receiver, assets)`. The strategy contract, holding the reserves, will transfer `assets` amount of sovaBTC to the `receiver`. We assume the strategy always maintains enough sovaBTC liquidity for redemptions (it may achieve this by converting some of the other collaterals behind the scenes or via admin operations).
  4. Emit a `Withdraw(msg.sender, receiver, owner, assets, shares)` event as per ERC-4626.

- *Withdraw (specify assets):* Alternatively, the user could call `withdraw(desiredSovaAmount, receiver, owner)`. In this case, we compute the shares to burn as `shares = convertToShares(desiredSovaAmount)`, then perform the same steps as above. The conversion would use the current price per share:

```
uint256 desiredAssets = ...; // sovaBTC amount requested (in sats)
uint256 underlyingScaled = desiredAssets * (10**(18 - 8)); // upscale to
18-dec
uint256 currentPrice = priceOracle.getCurrentPrice();
uint256 shares = (underlyingScaled * 1e18) / currentPrice;
```

We'd likely round up the required shares to ensure sufficient burn (avoiding leaving 1 sat worth due to division rounding).

Always Redeem in sovaBTC: Note that in both cases, the asset being delivered to the user is sovaBTC. The vault does *not* offer redemption in the originally deposited token, by design. This simplifies the user experience to a single predictable output and centralizes the conversion risk within the strategy (the protocol manages the various BTC collaterals internally).

- **totalAssets():** Reports the total value of assets under management in terms of sovaBTC. We can implement this in two ways, for transparency:
- **Oracle-based:** `totalAssets = (totalSupply of shares * pricePerShare) / 1e18`. Since pricePerShare is 18-dec and shares are 18-dec, this yields a value in 18-dec which we then downscale to 8-dec. This approach trusts the oracle's reported NAV completely.
- **Registry-sum-based:** Query the strategy's holdings of each collateral token and convert each to sovaBTC via the registry, summing them up. For example, the strategy could expose an array of supported collateral and balances. Then:

```
uint256 totalSova = 0;
for each collateral in registry:
    uint256 bal = IERC20(collateral).balanceOf(strategy);
    uint256 val = registry.getValueInUnderlying(collateral, bal);
    totalSova += val;
```

This gives total in sovaBTC base units (sats). We could then return `totalSova` (8-dec base) directly.

In practice, these two methods should agree **if the oracle is kept consistent with actual holdings**. We can use the oracle-based method for simplicity, but it is wise to double-check with the registry sum in critical situations (especially if conversion rates are adjusted or some collaterals deviate). For our implementation, we might implement `totalAssets()` using the oracle formula for efficiency, since it's a single multiplication and division.

- **convertToShares/assets:** As indicated above, these simply apply the current price per share. For example, `convertToShares(x)` multiplies `x * (10**(18-8)) * 1e18 / price` (to account for decimal difference and price scaling), whereas `convertToAssets(x)` does `x * price / 1e18 / (10**(18-8))`. These help integrators or UIs understand the exchange rate at any moment.

- **Access Control in Vault:** The vault's functions that need restriction are primarily administrative:

- Adding/removing collateral types in the registry (if the vault holds a reference to registry and exposes a method to add collateral, it should be `onlyRoles(PROTOCOL_ADMIN)`).
- Setting a new Strategy contract (if we allow upgrading the strategy via a setter on the vault, that function must be admin-only).
- Pausing deposits/withdrawals (if implemented) would be admin-only. For example, we could have `bool public paused; function setPaused(bool) external onlyRoles(PROTOCOL_ADMIN)` and then add `require(!paused, "Vault paused")` in deposit and withdraw functions.
- We will **not** restrict normal `deposit` or `redeem` functions – they are open to all users by design (no role check, no KYC required).

The vault contract will be initialized with the RoleManager address, and likely inherit from `RoleManaged` to use its modifiers. For instance, a snippet for an admin function:

```
function addCollateralToken(address token, uint256 rate, uint8 dec) external
onlyRoles(roleManager.PROTOCOL_ADMIN()) {
    registry.addCollateral(token, rate, dec);
}
```

This uses the RoleManager's stored role constant for PROTOCOL_ADMIN to gate the call [6]. The RoleManager contract itself is deployed once and holds the mapping of which addresses have that role.

- **Strategy interactions:** The vault primarily calls the strategy in two scenarios:
- after deposits (no immediate call needed besides sending tokens via Conduit; the strategy just receives funds),
- during withdrawals (calling `strategy.withdrawTo(sovaBTC, to, amount)` as described).

The strategy contract will have its own functions for the protocol admin to manage assets if needed (for example, an admin could call a function on strategy to swap WBTC for sovaBTC off-chain or on a DEX if that's part of the design, or to withdraw all assets to the admin in a shutdown scenario). Those would also be protected by roles. The strategy is constructed with knowledge of the vault or is set via `setVault` (the script calls `strategyImplementation.setSToken(vault)` which likely assigns the vault address) [10]. In our case, after deploying the vault proxy, we would call `strategy.setVault(vaultProxyAddress)` to establish the link. This ensures the strategy knows which vault is allowed to instruct it (we can also enforce in `withdrawTo` that `msg.sender` must be the vault).

- **Upgradeable pattern:** We will implement the Vault and Strategy as upgradeable. For a Transparent Proxy pattern, the actual Vault contract will have an initializer instead of a constructor (e.g. `initialize(address roleMgr, address oracle, address reg, address condu, address strat, address sovaAsset, uint8 sovaDecimals, string name, string symbol)`). This function will set up the state and call an internal `_initializeERC20(name, symbol, decimals)` for the share token. The proxies are deployed, pointing to the implementations. **After deployment**, the admin can upgrade them by:
- Deploying new implementation contracts (e.g. `MultiBTCVault_v2.sol`).
- Calling ProxyAdmin.upgrade(proxy, newImplementation) as the role-controlled admin.

The RoleManager isn't directly part of the proxy upgrade mechanism, but we ensure the ProxyAdmin is controlled by an address that has PROTOCOL_ADMIN, or if using UUPS, the Vault's `upgradeTo` is protected by `onlyRoles(PROTOCOL_ADMIN)`.

With the architecture defined, we can now proceed phase by phase through the implementation.

Phase 1: Initial Setup – Tokens and Core Contracts

Objective: Define and deploy the fundamental pieces: the existing tokens, role manager, oracle, and conduit, so that the system has a foundation to build on.

1. **Identify Existing Tokens:** List the BTC-based tokens to support:
2. **sovaBTC** – The primary asset for the vault (redeemable asset). It's assumed this token already exists in the Sova network. Confirm its address and decimals (should be 8). In tests, you might deploy a dummy ERC20 with 8 decimals named "sovaBTC".

3. **WBTC, tBTC, etc.** – For each BTC derivative, note its address on the target chain and decimals. Most wrapped BTC tokens have 8 decimals (e.g., WBTC is 8). Some versions of tBTC might use 18 decimals; ensure to fetch or configure the correct value. In a deployment script, you can use `IERC20(token).decimals()` or an environment config. For example, the Foundry script can map chain IDs to known addresses (as seen in FountFi's script logic) [11] .

4. **Deploy RoleManager:** If the protocol already has a RoleManager contract, you can reuse it; otherwise deploy a new one. This contract will be the cornerstone of access control. After deployment:

5. Determine the **admin addresses** (EOA or multisig) that should hold the `PROTOCOL_ADMIN` role. Typically, the deployer or a governance multisig gets this.
6. Call `grantRole(adminAddress, PROTOCOL_ADMIN)` for each admin (if not auto-granted). Some RoleManager implementations might automatically give the deployer all roles, but we verify this.
7. Also retrieve the role identifiers for later use (they might be constants in the contract or fetched via getters like `roleManager.PROTOCOL_ADMIN()`).

8. The RoleManager address will be passed to other contracts on construction, so note it down.

9. **Deploy PriceOracleReporter:** This oracle will start at a 1:1 price per share. In Foundry, you can deploy:

```
priceReporter = new PriceOracleReporter(
    1e18,              // initialPricePerShare = 1.000000000000000000 (18
decimals) [3]
    adminUpdater,      // address authorized to push updates (could be
PROTOCOL_ADMIN or a specific feeder EOA)
    100,               // maxDeviationPerTimePeriod = 100 bps (1%) –
example
    3600               // deviationTimePeriod = 1 hour – example
);
```

This sets NAV = 1.0 initially. The updater address can be one of your admin accounts (or a designated oracle bot's address). You can authorize multiple updaters: e.g. `priceReporter.setUpdater(secondaryAdmin, true)` [12] . Only these can call `priceReporter.update()`. Make sure the `PriceOracleReporter`'s owner (deployer) is set to

the governance or admin as needed (by default, the constructor's `_initializeOwner(msg.sender)` likely made the deployer the owner, so transfer ownership if necessary to a multisig). Keep the `priceReporter` address for use in vault initialization.

10. **Deploy Conduit:** The Conduit contract should be deployed with the address of the RoleManager:

```
conduit = new Conduit(address(roleManager));
```

This makes the Conduit aware of the central RoleManager for permission checks. By default, no vault is authorized to use it. After deploying the vault in Phase 2, we will need to **authorize the vault**:

11. Likely by granting the vault address a specific role that Conduit's `collectDeposit` requires. For example, if Conduit internally does `onlyRoles(roleManager.APPROVED_VAULT())` on `collectDeposit`, then:

```
roleManager.grantRole(vaultAddress, roleManager.APPROVED_VAULT());
```

would be executed by an admin. (If the RoleManager defines an explicit role for tRWA contracts, use that. In FountFi, Conduit is meant for "approved tRWA contracts" [5].)
12. Alternatively, the Conduit might maintain its own list (less likely since it inherits RoleManaged). We will assume RoleManager controls it.
13. We will perform this grant in the deployment script after the vault is deployed.

Also consider if the Conduit itself should have any special roles: e.g., PROTOCOL_ADMIN may need to be granted to the deployer on Conduit if not automatically (to allow calling `rescueERC20` or adding new approved vaults via roles). Check if `Conduit` constructor sets any roles for the deployer – if not, use RoleManager to grant your admin addresses whatever role controls Conduit (possibly PROTOCOL_ADMIN is enough since Conduit's admin functions use that [6]).

1. **Deploy MultiCollateralRegistry:** This contract likely requires the RoleManager and the base underlying asset:

```
mcRegistry = new MultiCollateralRegistry(address(roleManager),
sovaBTC_address);
```

The registry will use RoleManager for gating its `addCollateral`/`removeCollateral` functions (so only admins can add new tokens). Passing `sovaBTC` as a parameter indicates which token is considered the **underlying base** for conversions (so the registry might store that internally for reference).
2. After deployment, register the initial set of collateral tokens:

```
mcRegistry.addCollateral(sovaBTC_address, 1e18, 8);
mcRegistry.addCollateral(WBTC_address, 1e18, 8);
mcRegistry.addCollateral(tBTC_address, 1e18, 8);
```

```
// ... any others, e.g., cbBTC etc., each with conversionRate=1e18 and
their decimals [1] [13].
```

Explanation: We use `1e18` for each token to denote a 1:1 conversion rate to sovaBTC initially [1]. The third parameter is the token's decimals (all 8 in this example). Under the hood, the registry will likely store something like `(token -> {conversionRate: 1e18, decimals: 8})`. That means if asked, 1 WBTC (1*10^8 base units) = 1 * 1e18 * (10^8 / 10^8) = 1*10^8 sovaBTC base units (i.e., 1 sovaBTC). If a token had 18 decimals, say an ERC20 that represents BTC with 18 decimals, we would still put conversionRate 1e18 but decimals 18, so that 1 *10^18 units = 1e18 * (10^8/10^18) = 1e8 sovaBTC units. This consistent handling is done internally by `getValueInUnderlying`.

3. The registry might also have a function to mark an asset as a valid system asset in a central Registry (the script calls `registry.setAsset(token, decimals)` for each collateral [14]; this suggests there is a global Registry contract separate from MultiCollateralRegistry, but if not using a global one, we can ignore that step. If a global registry exists in the FountFi system, call those as needed for integration, but core functionality doesn't require it).
4. Only PROTOCOL_ADMIN can call `addCollateral`, so our script or deployment transaction doing this should be executed under an admin role (in Foundry script, since we deploy everything in one go as deployer, we'll likely have given deployer PROTOCOL_ADMIN at RoleManager deployment time, so it's allowed).

At the end of Phase 1, we have: - `roleManager` (with admin roles set up), - `priceOracleReporter` (initialized to 1.0 NAV), - `conduit` (ready to authorize vaults), - `multiCollateralRegistry` (populated with sovaBTC and other tokens).

These will be referenced when deploying the vault and strategy next.

Phase 2: Vault and Strategy Core Implementation

Objective: Implement and deploy the Multi-BTC vault (ERC-4626 share token) and its paired strategy, leveraging the components from Phase 1. This includes writing the Solidity code for these contracts following best practices and integrating RoleManager and upgradeability.

A. Vault Implementation (ERC-4626, upgradeable):

1. **Contract Skeleton:** Create a contract `MultiBTCVault` (in code, perhaps `contracts/vaults/MultiBTCVault.sol`). Make it inherit from:
2. `ERC20` (for the share token functionality; use OpenZeppelin's implementation but we will override `decimals()` to return 18).
3. `IERC4626` (you can use OpenZeppelin's ERC4626 as a base if single-asset, but since we extend to multi-asset deposit, you might implement from scratch or extend OZ and override some functions).
4. `RoleManaged` (to integrate RoleManager; typically this is an abstract that stores the RoleManager address and provides `onlyRoles` modifier).

Additionally, include state variables for: - `IERC20 public immutable underlying;` – sovaBTC token contract. - `uint8 private immutable underlyingDecimals;` - `IPriceOracleReporter public`

`priceOracle;` - `IMultiCollateralRegistry public registry;` - `IConduit public conduit;` - `IMultiCollateralStrategy public strategy;`

(If using proxies, these won't be "immutable" but rather storage variables set in initialize; mark them as upgradeable storage, not `immutable` in that case.)

Also possibly a `bool initialized;` if using a custom initializer pattern (or use OpenZeppelin's Initializable).

1. **Initializer:** Write an `initialize(...)` function (if using proxy) or a constructor (if not proxy) that sets up the contract:
2. Call `__RoleManaged_init(roleManagerAddress)` if such a function exists, or simply store the RoleManager address in a state variable.
3. Store the addresses for oracle, registry, conduit, strategy, underlying, and underlyingDecimals.
4. Initialize the ERC20 name/symbol/decimals. With OZ's InitializableERC20, you might call `_initialize(string memory name, string memory symbol, uint8 decimals)`. For example:

```
function initialize(address _roleManager, address _priceOracle, address
_registry, address _conduit, address _strategy, address _sova, uint8
_sovaDec) external initializer {
    __RoleManaged_init(_roleManager);
    __ERC20_init("Multi-Collateral Bitcoin Vault", "mcBTC");
    // we will override decimals() to 18, so no need to pass it if using OZ
ERC20 which defaults to 18 or allow __ERC20_init to set 18.
    priceOracle = IPriceOracleReporter(_priceOracle);
    registry = IMultiCollateralRegistry(_registry);
    conduit = IConduit(_conduit);
    strategy = IMultiCollateralStrategy(_strategy);
    underlying = IERC20(_sova);
    underlyingDecimals = _sovaDec;
    // maybe set an initial approval: ensure strategy can pull underlying
from vault if needed, but since vault doesn't hold underlying, maybe not
necessary.
}
```

   Note: The vault itself doesn't need to hold tokens, so it may not require allowances. The strategy will handle underlying, and the Conduit handles user->strategy transfers.

5. If not using a proxy (not recommended here), this would be in the constructor instead.
6. **ERC-20 Overrides:** Override `decimals()` to return 18 for the share token. This ensures any external interface knows mcBTC has 18 decimals.

7. **Depositing Function:** Implement `deposit(address token, uint256 amount, address receiver)`. We described the logic in **Vault Internal Logic Highlights** above. Key points:

8. Use `registry.getValueInUnderlying(token, amount)` to get the sovaBTC-value (in satoshis).
9. Use `priceOracle.getCurrentPrice()` for the current price per share (or decode `priceOracle.report()` bytes).
10. Compute shares to mint, as shown in the snippet, carefully handling precision. Use safe math or standard solidity since 0.8 has overflow checks (the multiplications and divisions with 1e18 are fine as long as values aren't extreme; they won't be beyond 1e18 * total supply likely).
11. Mint the shares to the receiver.
12. Call `conduit.collectDeposit(token, msg.sender, address(strategy), amount)` and check return value or use `.call` to handle a potential failure. (If Conduit returns false on failure, we require true; if it reverts on failure, wrap in try/catch if needed or just let it revert.)
13. Emit events. The ERC4626 standard `Deposit` event normally includes only one asset type, but here since multiple asset types are possible, we might emit a custom event that includes the token address. For simplicity, you can emit something like:

    ```
    event Deposit(address indexed caller, address indexed owner, address indexed token, uint256 tokenAmount, uint256 sharesMinted);
    ```

    This is a slight extension of the standard event to record which token was deposited.

14. **Permissions:** The deposit function is `external` and has no `onlyRoles` – it's open to all users. However, you may want to add `whenNotPaused` if a pause mechanism is implemented.

15. **Withdrawal/Redemption Functions:** Implement `redeem(uint256 shares, address receiver, address owner)` and `withdraw(uint256 assets, address receiver, address owner)`. Key points:

16. If `owner != msg.sender`, require `allowance(owner, msg.sender) >= shares` (ERC20 allowance mechanic) then `_spendAllowance(owner, msg.sender, shares)` before burning (OpenZeppelin ERC4626 handles this in their implementation of `redeem`, but if custom, do it manually).
17. Use the price oracle to compute underlying owed as shown above. Be mindful of rounding: it's generally okay to round down on withdraw (so the user gets slightly less sovaBTC if not exact), or track any remainder.
18. Burn the shares from the owner.
19. Call `strategy.withdrawTo(sovaAsset, receiver, assetsOwed)`. The strategy should then transfer that amount of sovaBTC to the receiver.
    - Ensure `assetsOwed` is available: you might do a safety check like `require(IERC20(sovaAsset).balanceOf(strategy) >= assetsOwed, "Insufficient liquidity");` to avoid a situation where strategy can't pay (though if that happens, the withdrawal fails – at least it won't silently underpay).
20. Emit the standard `Withdraw` event. (Standard ERC4626 Withdraw event has: caller, receiver, owner, assets, shares).

21. Like deposit, these functions are permissionless for users (no role check), but can be paused via circuit-breaker if needed.

22. **totalAssets and Conversions:** Implement `totalAssets()`, `convertToShares()`, and `convertToAssets()` using the oracle and stored values as discussed.

23. `totalAssets()` could simply do:

```
uint256 price = priceOracle.getCurrentPrice();
uint256 supply = totalSupply();
// both price and supply are 1e18-scaled relative to underlying (though
supply is just count with 18-dec).
// Compute in 18-dec and then scale down:
uint256 totalUnderlyingScaled = (supply * price) / 1e18;
uint256 totalUnderlying = totalUnderlyingScaled / (10**(18 -
underlyingDecimals));
return totalUnderlying;
```

    This returns the total sovaBTC in base units. (If you prefer to return in 18-dec scaled units to conform to some interface, you can, but usually totalAssets is expected in actual underlying units).

24. `convertToShares(assets)`:

```
// assets is given in sovaBTC base units (8-dec)
uint256 scaled = assets * (10**(18 - underlyingDecimals));
uint256 price = priceOracle.getCurrentPrice();
return (scaled * 1e18) / price;
```

25. `convertToAssets(shares)`:

```
uint256 price = priceOracle.getCurrentPrice();
uint256 underlyingScaled = (shares * price) / 1e18;
return underlyingScaled / (10**(18 - underlyingDecimals));
```

26. These allow external callers (like front-end or integration contracts) to determine rates easily. Also, our vault's internal deposit/withdraw logic can call these for cleanliness (e.g., use `convertToShares` in deposit after computing underlying value).

27. Note: You might also implement `maxDeposit`, `maxWithdraw` as unlimited for users (or limited by some cap or by liquidity). For example, `maxWithdraw(user)` should calculate the max sovaBTC that user can redeem given their share balance and current price, and also ensure strategy has that much; typically it would be all their assets if strategy liquidity is full. If the strategy might not have enough sovaBTC, an additional check could reduce maxWithdraw, but unless we track that on-chain, we assume full liquidity or rely on an off-chain check.

28. **Safety Hooks (optional):** If time, implement reentrancy guard (OpenZeppelin `ReentrancyGuard`) around deposit/redeem to be safe, as we are calling an external contract (strategy) during withdraw, which could possibly call back into vault (though our strategy is under our control and shouldn't reenter, but best practice). The FountFi audit noticed reentrancy in tRWA withdrawal hooks [15] – we will avoid such complexity (no user-supplied hooks in our design), but a simple `nonReentrant` modifier on deposit and withdraw won't hurt.

29. **Upgradeability:** Since this vault is upgradeable, make sure:

30. You use the `initializer` modifier on initialize and call initializers of parents (ERC20 doesn't have one by default, but OZ provides ERC20Upgradeable which has `__ERC20_init`).
31. Add a storage gap: e.g. `uint256[50] private __gap;` at the end of the contract to allow adding new state in the future without shifting existing storage.
32. The contract's storage layout must remain consistent on upgrades (hence, do not change the order of state vars).
33. Only the ProxyAdmin (held by PROTOCOL_ADMIN) or the vault's own upgrade function (if UUPS) can perform upgrades. We might rely on ProxyAdmin for simplicity.

B. Strategy Implementation (MultiCollateralStrategy):

The strategy will be a simpler contract that primarily holds tokens and allows the vault to withdraw sovaBTC for redemptions. We can model it on the FountFi `SimpleMultiCollateralStrategy`:

1. **Contract Setup:** Create `MultiCollateralStrategy` that inherits `RoleManaged` (to use RoleManager roles) and possibly Ownable or Initializable if needed. Key state:
2. `address public vault;` (the associated vault contract, a.k.a sToken in the script) [10].
3. `address public underlying;` (sovaBTC address) and `uint8 underlyingDecimals;` – to know which token is primary for redemption.
4. `IMultiCollateralRegistry public registry;` – to reference conversion rates if needed.
5. Maybe track RoleManager via RoleManaged (so it has access to `onlyRoles`).

6. Could also track an admin or manager address if needed (the script passes `MANAGER_1` to the strategy constructor [16] – likely that sets some admin who can operate the strategy or call certain functions like off-chain withdrawals).

7. **Initializer/Constructor:** The script shows an example:

```
strategy = new SimpleMultiCollateralStrategy(sovaBTC, sovaDecimals,
address(mcRegistry), MANAGER_1);
```

   So likely the strategy's constructor or init takes (address underlying, uint8 underlyingDec, address registry, address manager). We will do similarly:

8. Store `underlying` and `registry` and `underlyingDecimals`.
9. Perhaps set `vault = address(0)` initially; it will be set by vault deployment flow (vault calls `strategy.setVault(vaultAddr)` after creation).

10. Store `manager` or admin if needed (maybe for off-chain handling; but we might manage admin purely via RoleManager roles instead).
11. If RoleManager is used, it's passed via RoleManaged parent (e.g. `__RoleManaged_init(roleManagerAddress)` here as well).

12. Example:

```
function initialize(address _roleManager, address _underlying, uint8
_underDec, address _registry) external initializer {
    __RoleManaged_init(_roleManager);
    underlying = _underlying;
    underlyingDecimals = _underDec;
    registry = IMultiCollateralRegistry(_registry);
    vault = address(0);
}
```

    Then maybe assign manager roles: Instead of taking a manager address as param, we can rely on RoleManager's roles for controlling privileged operations. But if the strategy needs a distinct off-chain controller (like an address that calls swap or triggers yield distribution), we could incorporate that. The FountFi approach gave the strategy contract itself a STRATEGY_OPERATOR role [7], and maybe the `MANAGER_1` was set as an internal admin in strategy (perhaps the strategy inherits Ownable and sets owner = MANAGER_1 so that off-chain manager can call certain functions). We can incorporate an Ownable or just rely on RoleManager roles directly for controlling strategy functions. For simplicity, we might designate PROTOCOL_ADMIN to manage strategy as well, and possibly allow a specific role for operating it if necessary.

13. **Asset Holding:** The strategy will simply hold any tokens sent to it. We ensure it can receive ERC20 transfers (no special code needed for that).

14. **Withdraw Function:** Implement `withdrawTo(address asset, address to, uint256 amount)`:

15. `require(msg.sender == vault, "Unauthorized");` so only the vault can call it.
16. If `asset == underlying (sovaBTC)`: transfer `amount` of sovaBTC from strategy to `to`.
17. Else if we wanted to allow vault to pull other assets (perhaps during migrations), we could handle other tokens similarly. But normally vault will only call for sovaBTC on user redemption.
18. Use `SafeTransferLib.safeTransfer(IERC20(asset), to, amount)` for safety (from Solady or OZ).
19. This function doesn't need to compute anything, it just moves tokens.

20. If the strategy doesn't have enough of `asset`, the transfer will fail (SafeTransferLib will revert if transfer returns false). The vault should have already checked liquidity ideally.

21. **setVault Function:** Implement `setVault(address vaultAddr)`:

22. Only allow if not set yet (or if we plan to upgrade vault, maybe allow changing, but that could be risky). Perhaps restrict to `onlyRoles(PROTOCOL_ADMIN)` so an admin can re-point the strategy to a new vault if migrating.
23. `vault = vaultAddr;`
24. Optionally, you might give the vault the STRATEGY_OPERATOR role or some trust. However, since we already restrict withdrawTo to vault, that's sufficient. The RoleManager's STRATEGY_OPERATOR role is more relevant if strategy itself calls other contracts or performs privileged actions on vault or elsewhere.

25. In FountFi script, after deploying vault they called `strategyImplementation.setSToken(vault)` [10] – that corresponds to setting the vault address.

26. **Other Strategy Functions:** Depending on needs, you might implement:

27. `rebalance()` or `swapCollateral(address tokenFrom, address tokenTo, uint256 amount)` – for admins to convert some tokens to sovaBTC. These would be admin-only (use `onlyRoles(PROTOCOL_ADMIN)` or a specific operator role). They could integrate with DEX or bridge, but detailing that is beyond scope. We can note that out-of-band processes will handle ensuring the strategy accumulates yield and has liquidity.
28. `reportHoldings()` – possibly to assist oracle updates: e.g., returning total value of assets held (but since oracle is off-chain feed, it likely doesn't need on-chain reporting).

29. For simplicity, we won't implement on-chain investment logic; the strategy is a passive holder in this guide.

30. **Access Control & Roles in Strategy:**

31. The strategy will be deployed by the deployer (admin). We grant the strategy contract the `STRATEGY_OPERATOR` role in RoleManager [7], because perhaps the strategy might need to call certain protocol functions that require that (e.g., if strategy could call vault for something or interact with registry).
32. If the strategy includes admin functions like rebalancing, we protect them with `onlyRoles(PROTOCOL_ADMIN)` or a specialized role like `STRATEGY_ADMIN`.
33. By granting the strategy `STRATEGY_OPERATOR`, the protocol signals it can perform operations designated for strategies (for example, maybe calling an external protocol or receiving funds via Conduit if that required a role).
34. Additionally, **the vault might need a role** to call strategy's withdraw (depending on RoleManaged usage). We already restrict via `msg.sender == vault`, so no RoleManager check needed there. But if strategy had a RoleManaged check on withdrawTo, we'd ensure to grant vault some role like `VAULT_ROLE`.

35. To keep it simple: use address comparison for vault, and RoleManager for human/admin permissions.

36. **Upgradeability:** Similarly, the strategy is upgradeable:

37. Use an initializer.
38. Have a storage gap.
39. Upgrades would be done via proxy admin calling upgrade. The vault can remain pointed to the strategy proxy address. If we ever upgrade strategy logic, the vault's `strategy` address stays the same (proxy), only the implementation behind it changes.
40. If we want to entirely replace the strategy (deploy new proxy), we'd have to do a migration of funds from old to new and then update the vault's stored strategy address to the new one. That's more complex (requires admin procedure). With an upgradeable proxy for strategy, we likely won't need to replace the address, just upgrade logic in place.

C. Deployment of Vault & Strategy:

Now that we have the Vault and Strategy code, we deploy them (likely via proxies):

1. **Deploy Implementation Contracts:**
2. `vaultImpl = new MultiBTCVault();` (if using OpenZeppelin Upgrades plugin, you might skip direct deploy and use proxy factories; but with Foundry, we can deploy and then proxy manually or use OZ's TransparentUpgradeableProxy contract).
3. `strategyImpl = new MultiCollateralStrategy();` These are the logic contracts.
4. **Deploy Proxies:**
5. If using Transparent proxies: deploy a `TransparentUpgradeableProxy` for each:

   ```
   proxyVault = new TransparentUpgradeableProxy(address(vaultImpl),
   address(proxyAdmin), "");
   proxyStrategy = new TransparentUpgradeableProxy(address(strategyImpl),
   address(proxyAdmin), "");
   ```

   Here `proxyAdmin` is an instance of OpenZeppelin's `ProxyAdmin` that is controlled by our deployer (we could deploy ProxyAdmin and then transfer ownership to the governance multisig or keep deployer as admin for now). We pass empty `""` init data for now, meaning we will call initialize separately.
6. Alternatively, use the `ERC1967Proxy` contract (which can take an initializer data in its constructor). For example:

   ```
   bytes memory initData =
   abi.encodeWithSelector(MultiBTCVault.initialize.selector, roleManagerAddr,
   priceOracleAddr, registryAddr, conduitAddr, /* strategy proxy address
   placeholder */, sovaAddr, sovaDecimals);
   proxyVault = new ERC1967Proxy(address(vaultImpl), initData);
   ```

   But notice, the vault's initialize needs the strategy's address, and the strategy's initialize might not need the vault. To resolve circular dependency:

- We can deploy strategy proxy first (with strategy initialized without knowing vault).
- Then deploy vault proxy with the strategy's address.
- Finally call `strategy.setVault(vaultProxyAddress)` to complete the link. Because in our design the vault needs strategy address up front (to store it), and strategy needs vault address (to restrict calls). The script handled this by deploying vault (passing strategy address in constructor), then calling `strategy.setSToken(vault)` afterwards [10]. We will mimic that.

So: - **Deploy strategy proxy first:** initialize it with (roleManager, underlying=sovaBTC, underlyingDec, registry). The strategy doesn't need the vault at init. - **Deploy vault proxy next:** now include the `strategyProxy` address in its init data. - Then call `strategyProxy.setVault(vaultProxyAddr)` in a separate transaction (or script step). - Ensure to set ProxyAdmin ownership appropriately (the ProxyAdmin should be owned by PROTOCOL_ADMIN role holder).

7. **Initialize Strategy Proxy:** Call `strategy.initialize(roleManagerAddr, sovaBTC, 8, registryAddr)` via the proxy (or embed in constructor of ERC1967Proxy).

8. If strategy has an admin/manager address parameter (like `MANAGER_1` in script), you could include that too, e.g. `initialize(roleManager, sova, 8, registry, managerAddr)`. Or simply rely on RoleManager roles for management.

9. After init, perhaps grant the strategy any required roles:

   - `roleManager.grantRole(strategyProxyAddr, roleManager.STRATEGY_OPERATOR())` [7] – this allows the strategy contract to perform any actions that require STRATEGY_OPERATOR role in the system.
   - If there's a special role for being allowed to pull from Conduit (though vault handles Conduit calls, not strategy).
   - If strategy might call registry to add assets on the fly (not likely), could consider roles, but probably not needed.

10. **Initialize Vault Proxy:** Prepare `initData` for vault:

11. Use the addresses from Phase 1 and the new strategy proxy:

```
vault.init(roleManagerAddr, priceOracleAddr, registryAddr, conduitAddr,
strategyProxyAddr, sovaBTCAddr, 8);
```

12. This sets everything up. After this call, the vault is live.
13. The vault will hold references to the strategy proxy (so it will call that on withdraw).
14. **Important:** Now authorize the vault in Conduit:
    - `roleManager.grantRole(vaultProxyAddr, roleManager.APPROVED_VAULT())` (assuming such a role exists for Conduit to check). This makes the vault an approved caller of `collectDeposit` [5].

- If Conduit instead has its own list, call e.g. `conduit.approveVault(vaultProxyAddr)` if that function exists (not in the snippet, so likely the role approach).

15. Also possibly register the strategy in some global registry if one exists: the script called `registry.setStrategy(strategyAddr, true)` [17], which suggests a global Registry that tracks active strategies (maybe for the frontend or risk checks). If our system has such, we do the same. Otherwise skip.

16. **Configure Strategy Vault Link:** Call `strategy.setVault(vaultProxyAddr)` now. This finalizes the circular relationship:

17. Only an admin (PROTOCOL_ADMIN) should call this, or we can allow `setVault` to be callable by the vault itself if it passes a check. But simpler: we call it in the deployment script as deployer (who is admin).

18. After this, the strategy knows its vault, and vault knows its strategy. The deployment is complete.

19. **Post-Deployment Access Control:**

20. Confirm the RoleManager roles:
    - The deployer or intended governance has PROTOCOL_ADMIN (so they can manage).
    - Strategy has STRATEGY_OPERATOR [7].
    - Vault might not need a special role in RoleManager, but it's now approved in Conduit via role.
    - If there is a `VAULT_OPERATOR` role and if vault needs to call certain restricted actions on strategy (not in our simple design), we'd grant vault that. But we limited vault's influence to strategy via `msg.sender == vault` checks.
21. If there's any other role like a PAUSER, and we want say the admin or a sentinel address to be able to pause, grant that role appropriately and set up pause logic in vault/strategy if implemented (e.g., strategy might have an emergency withdrawal function for admin to pull all funds to a safe address – guard it with PROTOCOL_ADMIN).

At the end of Phase 2, we have a functioning vault and strategy deployed: - **MultiBTCVault (mcBTC)** proxy at some address – users will interact with this for deposits and withdrawals. - **MultiCollateralStrategy** proxy holding no funds yet (until users deposit). - All supporting contracts (registry, oracle, conduit, roleManager) wired in. - Roles configured so that: - Only admins can add collateral types or upgrade contracts. - Only the vault can call the strategy to withdraw funds. - Only the vault can call Conduit to move user funds. - Only authorized oracle updaters can change NAV. - Users can freely deposit/withdraw when not paused.

Before moving on, it's wise to write unit tests (with Foundry or Hardhat) for deposit and withdraw flows to ensure the conversions and share accounting work as expected (especially with different decimals). Also test edge cases: depositing the smallest unit (1 satoshi), withdrawing all shares, multiple deposits and NAV update in between, etc.

Phase 3: Oracle Integration and NAV Updates

Objective: Integrate the PriceOracleReporter feed into the operations and outline how NAV updates occur to reflect yield or losses. This phase is mostly about operational setup rather than coding, since the vault already calls the oracle, but it's crucial to understand how to use the oracle effectively.

1. **Connecting Oracle in Vault/Strategy:** In the vault's storage, we have `priceOracle` set. All conversions use `priceOracle.getCurrentPrice()`. Ensure that this is indeed reading from the PriceOracleReporter contract deployed in Phase 1. If the oracle uses a bytes return (via `report()`), and we only have IReporter interface, we might need to decode. Our code assumed `getCurrentPrice()` exists (since in the FountFi implementation, `PriceOracleReporter` has a public function for current price). If not, we can do:

   ```
   uint256 currentPrice;
   bytes memory priceBytes = IPriceOracleReporter(priceOracle).report();
   // decode to uint256
   currentPrice = abi.decode(priceBytes, (uint256));
   ```

   But given we have the concrete type, `PriceOracleReporter(priceOracle).getCurrentPrice()` works too (it's public) [18]. We should use whichever is simpler.

2. **Initial NAV Setup:** At deployment, the PriceOracleReporter was set to 1e18 (1.0). That means initially 1 share = 1 sovaBTC. We should double-check this by perhaps calling `vault.convertToAssets(1e18)` after deployment, expecting ~1e8 (which would be 1 sovaBTC in sats). This confirms the wiring.

3. **Admin Oracle Updates:** Document how an admin will update NAV:

4. Suppose after some off-chain yield or interest, the vault's holdings increased by 5%. Instead of dealing with tokens on-chain, the admin can simply report a new price per share that is 1.05e18.
5. The admin (with address authorized as updater) calls `priceOracleReporter.update(1_05000000000000000, "Q2 yield")`. The `source` string can be a short description or identifier.
6. The PriceOracleReporter will emit an event `PricePerShareUpdated(round++, newPrice, oldPrice, source)` [19] [20], and internally it may start a gradual transition if the change exceeds the allowed deviation per period. In our example, if max deviation is 1% per hour, a 5% jump triggers a transition: the oracle will initially report maybe 1% now and gradually move to 5% over time. However, because our vault uses `getCurrentPrice()`, it automatically gets the intermediate price. So share conversions will smoothly increase as the price transitions to target.
7. If an immediate jump is needed (e.g., no arbitrage concern), the admin can set a high deviation or use `forceCompleteTransition()` (owner-only) to apply instantly [21].

8. **Staleness:** The vault code doesn't itself check if the oracle price is fresh or stale. The PriceOracleReporter doesn't have a built-in staleness check either (in audit they recommended

adding one) [22]. We rely on governance process to update or pause if oracle data is old. If needed, we could add in vault a requirement that `block.timestamp - lastOracleUpdate < X` for deposits to avoid accepting deposits on stale price (to prevent someone exploiting an outdated low price to deposit cheap). But that adds complexity and is optional. At minimum, have a monitoring off-chain that the oracle is updated regularly if yield is expected.

9. **NAV and User Interactions:** With the oracle integrated:

10. If NAV increases, **existing share holders gain value** (each share is worth more sovaBTC). New deposits will get fewer shares per token deposited (as shown in conversion math), so they buy in at the higher NAV fairly.
11. If NAV were to decrease (e.g., if an off-chain loss happened and oracle updates price downwards), each share loses value. Withdrawals after that point will yield fewer sovaBTC per share. Depositors who come in after the loss get a better price (more shares for their input), again fairly reflecting the loss. The system basically socializes gains or losses to all shares via price updates.
12. The oracle thus needs to be updated whenever a material change in the vault's true asset value occurs. In a pure crypto context, if the strategy just holds the tokens, the NAV should remain ~1 (assuming all tokens remain pegged 1:1 to BTC). Minor fluctuations or defaults of a collateral could be handled by changing its conversion rate in the registry (for instance, if one derivative depegs to 0.9 BTC, an admin might change its conversionRate to 0.9e18). But such changes ideally should also reflect in the NAV if not already accounted.

13. In summary, **for off-chain yields**: update PriceOracleReporter; **for on-chain collateral value changes**: update MultiCollateralRegistry (and possibly PriceOracleReporter if overall vault value changes significantly).

14. **Example Code Usage:** An example of pulling the price in vault:

```
uint256 price = PriceOracleReporter(priceOracle).getCurrentPrice();
```

which gives a `uint256` 18-dec. Or using the interface:

```
bytes memory data = priceOracle.report();
uint256 price = abi.decode(data, (uint256));
```

We have already integrated this in our deposit/withdraw logic.

15. **Testing Price Updates:** We should test scenario:

16. User A deposits 1 WBTC (gets ~1 share initially).
17. Oracle updated to 1.1 (10% gain).
18. User B deposits 1 WBTC after update (they should get ~0.909 shares).
19. User A withdraws 1 share (they should get 1.1 WBTC worth in sovaBTC).
20. This ensures the math is consistent (User A gained from the NAV increase).

Another test: if oracle reduces price below 1, share conversion works similarly (though hopefully that's rare in our case unless something bad happened).

1. **Centralization Note:** It's worth mentioning in documentation that the **oracle is centralized** – this is acceptable in this design (and noted in the audit as a centralization risk [23]). The trade-off is flexibility in managing real-world yields and multi-token parity. The system assumes trust in the updater to report honestly; a compromised oracle updater could maliciously set an incorrect price (though our `maxDeviation` and time-based limits mitigate how fast they can shift it). In production, consider multisig control or time delays for oracle updates if possible.

2. **Fail-safe:** If the oracle is not updated for a long time but collateral values remain roughly 1:1, the vault still functions (it will just treat NAV as last known value). If yields are supposed to accrue and oracle isn't updated, share value stays stale (meaning effectively yields aren't realized on-chain – effectively the vault is under-reporting value). It's not catastrophic unless someone exploits timing. So it's mainly a matter of proper operation to update regularly.

In summary, Phase 3 ensures the vault is correctly using the PriceOracleReporter and that procedures are in place for admins to update NAV through it, keeping the vault's accounting in sync with reality.

Phase 4: Administration, Control & Upgradeability

Objective: Enumerate the administrative functions and controls in the system, and ensure the upgradeable architecture and role permissions are clearly defined for maintenance.

Admin Controls Provided:

- **Adding New Collateral Types:** The protocol admin (RoleManager's PROTOCOL_ADMIN) can add support for another BTC derivative token post-deployment. To do so, they would:
- Deploy or identify the token's address and decimals.
- If using on-chain governance, propose a transaction or if using a multisig, execute:

```
mcRegistry.addCollateral(newToken, conversionRate, decimals);
```

  Usually `conversionRate` starts as 1e18 if the token is intended to equal 1 BTC [1]. If the token is known to trade at a slight discount or premium, admin could set accordingly.
- After adding, also possibly call `roleManager.grantRole(vaultAddr, roleManager.APPROVED_VAULT())` for vault if not already (the vault is already approved from before, so not needed again) – but if a *new vault* was ever added to system, that step would matter.
- Now the vault automatically can accept that token in `deposit` (the registry check will pass).

- Removing a collateral or disabling it could be done via a function (not specified, but we could implement `removeCollateral(token)` admin-only that marks it invalid so deposits reject it going forward).

- **Oracle Price Updates:** As described, authorized updaters (could be a role for oracle or just specific addresses) will periodically adjust `PriceOracleReporter`. The owner of PriceOracleReporter

(likely PROTOCOL_ADMIN or a trusted account) can also call `setMaxDeviation` to tune how fast price can change [24], or `forceCompleteTransition` to skip the gradual phase in emergencies [25]. Those functions are `onlyOwner` in PriceOracleReporter, so effectively controlled by the admin who owns that contract.

- **Emergency Pause:** If we included a pause feature:

- We would have `bool public paused` in vault (and possibly strategy).
- `function pause(bool _state) external onlyRoles(PROTOCOL_ADMIN)` to set it.
- Then `deposit` and `redeem` functions start with `require(!paused, "Paused");`.
- This allows halting user operations during emergencies (e.g., oracle failure or extreme market event). Since no KYC or other gate exists, pause is the main emergency brake.
- The Conduit could also have a pause (not indicated in snippet, but we can assume not necessary if vault pausing suffices).

- If implemented, ensure to test that pausing stops new deposits and withdrawals, and unpausing restores functionality.

- **Upgrading Core Contracts:** Using Transparent proxies means:

- The ProxyAdmin contract's owner (the deployer or set governance) has the authority to change the implementation addresses of the vault or strategy proxies.
- To upgrade:
    1. Write new implementation contract (e.g., `MultiBTCVaultV2`) with improvements or fixes. Make sure to add new variables after the gap, etc.
    2. Deploy the new implementation to the network.
    3. Have the ProxyAdmin (controlled by PROTOCOL_ADMIN role effectively) call `upgrade(proxyVaultAddress, newVaultImplAddress)`. This will update the proxy to use the new code. Because Transparent proxies disallow the implementation contract from being called by users (to avoid clashing with ProxyAdmin), only the admin can do this.
    4. If an initialization step is needed (e.g., to set new variables), they would use `upgradeAndCall(...)` to call an initializer function in the same transaction.
    5. Do similar for strategy if needed.
    6. It's advisable to pause the vault during an upgrade to avoid user interactions in the middle of implementation switching.
- **Storage Safety:** The new implementation must maintain the same storage layout for all existing state. For example, if we add a new feature requiring a variable, we append it after the `__gap` or reduce the gap by one (ensuring indices align). Changing types or order of existing variables would corrupt data (e.g., swapping the registry and oracle addresses in storage by accident). Use tools or careful code review to ensure slot alignment.

- Because our vault and strategy are fairly straightforward in state (addresses and some small vars), this is manageable. We documented an array of 50 `__gap` slots to allow a number of new variables in the future.

- **Role Management:**

- The RoleManager contract allows adding/removing addresses from roles. For instance, if a new team member should have permission to update the oracle, an admin can call `priceOracleReporter.setUpdater(newUpdater, true)` (oracle-specific), or if we had a role for that in RoleManager (not really used for oracle in our design).
- If a new vault or strategy is deployed, we grant them roles accordingly.
- If a contract is compromised or deprecated, an admin can revoke its roles:

```
roleManager.revokeRole(oldStrategyAddr, roleManager.STRATEGY_OPERATOR());
```

  This, along with perhaps pausing, could freeze an old strategy's actions.

- RoleManager may have an event log for these changes, aiding transparency.

- **No User-Specific Controls:** Emphasize that there are **no per-user whitelist or blacklist functions**. The admin **cannot selectively block an individual user's deposits or withdrawals via the smart contract** (unless they pause the entire system). This aligns with "no KYC hooks" – the contracts do not check any allowlist. All addresses are treated equally by the code.

- **Monitoring and Maintenance:** Admins should monitor:

- Collateral token health (if any token loses peg or has issues, consider removing it from allowed list and converting those holdings to sovaBTC if possible).
- Oracle updates (ensure periodic updates if yield expected; ensure no unauthorized updates).

- System metrics like totalAssets vs actual off-chain holdings, just to ensure no discrepancy.

- **Upgrading Conduit or RoleManager:** These are core infra as well:

- The RoleManager is usually not upgraded often (since it's simple mapping of roles), but if needed, that could be a new deployment – however, all RoleManaged contracts reference the original. Changing it would require redeploying everything or adding support for switching role manager (not trivial). It's best to get RoleManager right from the start.

- The Conduit might be upgradeable if deployed via a proxy (depending on implementation). If not, and a change is needed (say a bug fix), one might deploy a new Conduit and update vaults to use that for deposits (and get users to re-approve). This is cumbersome, so ideally Conduit is simple and robust. For now, we assume Conduit is fine as is (the functions are straightforward token transfer logic).

- **Interfaces and Documentation for Agents:** Because the developer using GPT-5 will be implementing this, we want to ensure all interfaces are clearly defined. We have given interface outlines above. Additionally, writing NatSpec comments in the code will help:

- E.g., `/// @notice Deposits a BTC token and mints vault shares. See {IMultiBTCVault-deposit}.`
- And for each complex math section, include comments as we've done to explain why scaling by 1e10 or 1e18, etc., so future maintainers or auditors (human or AI) understand the intent.

Summary of Roles and Permissions:

- **PROTOCOL_ADMIN (Gov MultiSig)** – Can add collaterals, pause/unpause, upgrade contracts (via ProxyAdmin control), manage roles, and directly own PriceOracleReporter (hence can add updaters or emergency update price). Essentially full control to manage the system's configuration.
- **Oracle Updater (e.g. Yield Manager bot)** – Has permission (via PriceOracleReporter's `authorizedUpdaters`) to call `update(price)` on the oracle. This can be the same as PROTOCOL_ADMIN or a separate dedicated role. If separate, ensure the updater cannot do other admin actions (least privilege).
- **STRATEGY_OPERATOR (Strategy contract)** – This role is granted to the **Strategy contract instance**. It may allow the strategy to call certain privileged functions, though in our current design we didn't need the strategy to call anything externally that requires a role. However, FountFi likely envisioned strategy calling out (like interacting with external protocols) and needing that role as a gate. In our case, we still grant it for consistency. The role essentially tells the system "this strategy is trusted to operate within the protocol".
- **Approved Vault (Conduit)** – The vault contract is granted whatever role or approval the Conduit requires for `collectDeposit`. This ties the vault into the Conduit's authorized list. Without it, any attempt to deposit will revert with an auth error in Conduit.

After Phase 4, we have a governance and maintenance plan for the vault. The developer should clearly separate **user pathways** (open access) from **admin pathways** (role-restricted) in the code.

Before final deployment, double-check each admin function for proper `onlyRoles` modifiers and each user function for no unintended restrictions. Also confirm that roles like PROTOCOL_ADMIN cannot accidentally be renounced or taken over (RoleManager should only let current admins manage roles).

Phase 5: Deployment & Foundry Testing/Script Usage

Objective: Use Foundry (Forge) to deploy the contracts and ensure everything is working. Provide a script and instructions on running it, as well as mention testing and coverage.

Deployment Script (Foundry): Create a script file at `script/DeployVault.s.sol` (Foundry convention for scripts). In this file, we'll write a `Script` contract to handle the deployment transactions. For mainnet/production deployment, you'd likely use a multi-step approach with environment variables for keys and addresses. For testing, you can simplify or hardcode some values.

Below is a pseudo-code for `DeployVault.s.sol`:

```
// script/DeployVault.s.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Script.sol";
import { RoleManager } from "../src/auth/RoleManager.sol";
import { PriceOracleReporter } from "../src/reporter/PriceOracleReporter.sol";
import { Conduit } from "../src/conduit/Conduit.sol";
```

```
import { MultiCollateralRegistry } from "../src/registry/
MultiCollateralRegistry.sol";
import { MultiBTCVault } from "../src/vaults/MultiBTCVault.sol";
import { MultiCollateralStrategy } from "../src/strategy/
MultiCollateralStrategy.sol";
import { TransparentUpgradeableProxy } from "../lib/OpenZeppelin/contracts/
proxy/transparent/TransparentUpgradeableProxy.sol";
import { ProxyAdmin } from "../lib/OpenZeppelin/contracts/proxy/transparent/
ProxyAdmin.sol";

contract DeployVaultScript is Script {
    // Config: replace with actual addresses if deploying to existing system
    address constant EXISTING_ROLE_MANAGER = address(0);  // set to nonzero to
skip deploying a new one
    address constant EXISTING_REGISTRY = address(0);
    address constant SOVABTC = 0x...;  // known sovaBTC address on network
    address constant WBTC = 0x...;
    address constant TBTC = 0x...;
    // ... add others as needed

    function run() external {
        uint256 deployerPrivKey = vm.envUint("PRIVATE_KEY");
        address deployer = vm.addr(deployerPrivKey);
        vm.startBroadcast(deployerPrivKey);

        // 1. Deploy or fetch RoleManager
        RoleManager roleManager;
        if (EXISTING_ROLE_MANAGER == address(0)) {
            roleManager = new RoleManager();
            // grant deployer as PROTOCOL_ADMIN if not automatic
            roleManager.grantRole(deployer, roleManager.PROTOCOL_ADMIN());
        } else {
            roleManager = RoleManager(EXISTING_ROLE_MANAGER);
        }

        // 2. Deploy Price Oracle Reporter (initial 1e18 price, allow deployer
as updater)
        PriceOracleReporter priceOracle = new PriceOracleReporter(
            1e18,
            deployer,   // authorize deployer or designated updater
            100,
            3600
        );

        // 3. Deploy Conduit
        Conduit conduit = new Conduit(address(roleManager));

        // 4. Deploy MultiCollateralRegistry
```

```
        MultiCollateralRegistry mcRegistry = new
MultiCollateralRegistry(address(roleManager), SOVABTC);
        // Add collaterals
        mcRegistry.addCollateral(SOVABTC, 1e18, 8);
        mcRegistry.addCollateral(WBTC, 1e18, 8);
        mcRegistry.addCollateral(TBTC, 1e18, 8);
        // ... add others, using 1e18 and correct decimals
        // Note: roleManager ensures only deployer (admin) can call this.

        // 5. Deploy ProxyAdmin for upgradeable proxies
        ProxyAdmin proxyAdmin = new ProxyAdmin();
        // Optionally, transfer ownership of ProxyAdmin to a multisig or keep
deployer for now:
        // proxyAdmin.transferOwnership(GOV_MULTISIG);

        // 6. Deploy Vault and Strategy implementations
        MultiBTCVault vaultImpl = new MultiBTCVault();
        MultiCollateralStrategy stratImpl = new MultiCollateralStrategy();

        // 7. Deploy Strategy proxy and initialize
        TransparentUpgradeableProxy stratProxy = new
TransparentUpgradeableProxy(
            address(stratImpl), address(proxyAdmin),
            abi.encodeWithSelector(
                MultiCollateralStrategy.initialize.selector,
                address(roleManager), SOVABTC, uint8(8), address(mcRegistry)
            )
        );
        MultiCollateralStrategy strategy =
MultiCollateralStrategy(address(stratProxy));
        // Grant strategy the STRATEGY_OPERATOR role
        roleManager.grantRole(address(strategy),
roleManager.STRATEGY_OPERATOR());

        // 8. Deploy Vault proxy and initialize
        TransparentUpgradeableProxy vaultProxy = new
TransparentUpgradeableProxy(
            address(vaultImpl), address(proxyAdmin),
            abi.encodeWithSelector(
                MultiBTCVault.initialize.selector,
                address(roleManager),
                address(priceOracle),
                address(mcRegistry),
                address(conduit),
                address(strategy),
                SOVABTC,
                uint8(8)  // sovaBTC decimals
            )
```

```
        );
        MultiBTCVault vault = MultiBTCVault(address(vaultProxy));

        // 9. Configure cross-links and permissions
        // Set vault in strategy
        strategy.setVault(address(vault));
        // Authorize vault to use Conduit

roleManager.grantRole(address(vault), /* roleManager.APPROVED_VAULT() or
appropriate */ bytes32(0));
        // (In actual code, replace bytes32(0) with the real role constant for
approved vaults if defined)

        vm.stopBroadcast();

        // Logging addresses
        console.log("Vault (mcBTC) deployed at:", address(vault));
        console.log("Strategy deployed at:", address(strategy));
        console.log("PriceOracleReporter deployed at:", address(priceOracle));
        console.log("MultiCollateralRegistry at:", address(mcRegistry));
        console.log("Conduit at:", address(conduit));
        console.log("RoleManager at:", address(roleManager));
        console.log("ProxyAdmin at:", address(proxyAdmin));
    }
}
```

This script (in a format similar to the FountFi deployment scripts) does the following: - Uses `vm.envUint("PRIVATE_KEY")` to fetch a private key from environment (so as not to hardcode secrets). You would supply this when running the script. - Broadcasts transactions with `startBroadcast`. - Logs important addresses at the end. (Foundry `console.log` will output to the console when running the script).

Running the Script: - To execute on a local development network or testnet: use the Forge CLI with appropriate flags. For example, to run on a fork or local node:

```
forge script script/DeployVault.s.sol:DeployVaultScript --fork-url http://
localhost:8545 --broadcast
```

Or to run on a testnet (Sepolia, for instance):

```
forge script script/DeployVault.s.sol:DeployVaultScript --rpc-url
$SEPOLIA_RPC_URL --broadcast --verify
```

including `--verify` if you want to verify contracts on Etherscan. - The script uses the `TransparentUpgradeableProxy` and `ProxyAdmin` from OpenZeppelin; ensure those are imported

correctly (the path in import may need adjustment to your project's structure or using OpenZeppelin installed via forge. - If any step fails, Forge will report the error. Common issues might be roleManager requiring admin for addCollateral – but we gave deployer that role, so it should pass.

Testing the Deployment: After running, you can verify: - The vault's name, symbol, and decimals via `vault.name()`, `vault.symbol()`, `vault.decimals()`. - The vault's initial totalAssets should be 0 (no deposits yet). - Try a test deposit on a local fork: - Impersonate a user, have them approve Conduit, call `vault.deposit(WBTC, x, user)`, then check their `vault.balanceOf(user)` and `strategy.balanceOf(WBTC)` etc. - Ensure the user's WBTC balance reduced, strategy's WBTC increased, and shares minted correctly. - Test redeem: call `vault.redeem(shares, user, user)` and see that user gets sovaBTC and shares burn. - Test price updates: simulate by calling `priceOracle.update(newPrice, "test")` as authorized updater and then see `vault.convertToAssets(1e18)` change accordingly.

Forge Build & Coverage: - Run `forge build` to compile all contracts. This should catch any compilation issues. - Write unit tests in `test/` directory (in Solidity or Foundry's fuzz style). For example, `test/MultiBTCVault.t.sol` with scenarios: - deposit/withdraw of one asset, - deposit with one asset, withdraw with another (shouldn't be allowed – user can't choose other asset), - multiple deposits, oracle update, then withdrawals, - role restrictions (e.g., try non-admin to add collateral, expect revert). - Execute `forge test -vvv` for verbose output on tests. - Use `forge coverage` to generate a coverage report. This will instrument the tests to see which lines of code executed. Aim for high coverage especially on critical math and role logic. - If some parts are hard to test (like Proxy upgrade flows), it's okay, but ensure deposit/withdraw and admin functions are covered. - Foundry's coverage output will show a summary; you can also use `forge coverage --report lcov` for an lcov info file which can be used with genhtml to create a web report. - Address any failing tests or uncovered scenarios. Common fixes might be adjusting for rounding, adding checks, etc.

Project Structure & Conventions: - Place contracts in appropriate directories (we used `src/vaults`, `src/strategy`, etc., adjust as per your repo). - Name scripts clearly (DeployVault.s.sol as requested). - Use consistent naming: we used "mcBTC" for the vault token symbol. You could also call it something like "Fount Satoshi Index" but mcBTC is clear and was used in the example script [26]. - Comments in code: ensure every function of the vault and strategy has a NatSpec comment explaining it. This helps any AI agent or human to understand without confusion.

By the end of Phase 5, the vault system should be fully deployed, with scripts to reproduce deployment and tests ensuring correctness. The single developer (GPT-5 agent) following this guide should have unambiguous instructions to implement and verify each component.

Conclusion: Following these phases, we achieve a **modular, upgradeable multi-collateral BTC vault** integrated into the FountFi framework. The system uses **8-decimal BTC assets** and **18-decimal shares** correctly, is **admin-controllable** (via RoleManager roles and oracle feed) yet **permissionless for users**, and follows **Foundry development conventions** for implementation and deployment (clean contract separation, script in `script/` directory, and standard Forge commands for build/test). By combining code examples and commentary as above, another agent or developer can readily implement and maintain the vault with clarity on each step of the process.

1 3 7 8 9 10 11 12 13 14 16 17 26 DeployMultiCollateralProduction.s.sol

https://github.com/SovaNetwork/fountfi-open/blob/26a6ab481d278f0c6acfe4c3f182b420e3c99e04/script/DeployMultiCollateralProduction.s.sol

2 18 19 20 PriceOracleReporter.sol

https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/src/reporter/PriceOracleReporter.sol

4 21 24 25 contract.PriceOracleReporter.md

https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/docs/src/src/reporter/PriceOracleReporter.sol/contract.PriceOracleReporter.md

5 6 contract.Conduit.md

https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/docs/src/src/conduit/Conduit.sol/contract.Conduit.md

15 22 23 AUDIT_OPUS.md

https://github.com/SovaNetwork/fountfi/blob/ddb94a5d9934618c92d2555b11e9e1390656c72e/audits/AUDIT_OPUS.md