

Kiln: Phase 2 Security Review

Auditors

Alireza Arjmand, Lead Security Researcher

Optimum, Lead Security Researcher

StErMi, Lead Security Researcher

Zigtur, Lead Security Researcher

Report prepared by: Lucas Goiriz

September 3, 2026

Contents

1 About Spearbit **4**

2 Introduction **4**

3 Risk classification **4**
3.1 Impact . . . 4
3.2 Likelihood . . . 4
3.3 Action required for severity levels . . . 4

4 Executive Summary **5**
4.1 Scope . . . 5

5 Overview **7**

6 Findings **8**
6.1 Medium Risk . . . 8
6.1.1 Inaccurate `isVehicleSector` masking allows to manipulate `totalAssets` . . . 8
6.1.2 Problems and side effects of the `QueryRedeemQueue.retrievable` behavior . . . 8
6.1.3 MultiVehicle's `totalAssets()` ignores sub-vehicle redeem fees, inflating share price and fees . . . 10
6.1.4 Queued redeem demand inflates child MV NAV, Parent MV deposits at understated share price 16
6.1.5 `requestWithdrawable` can modify `totalAssets` and let fulfillment have a better rate . . . 17
6.1.6 Automatic allocations to vehicles with deposit fees break share fairness . . . 30
6.2 Low Risk . . . 32
6.2.1 Improve the `VehicleLib.isCategory` validation for the `SingleAsset` category . . . 32
6.2.2 `BaseVehicle` functions `maxDeposit` and `maxRedeem` should return zero when the `create` function cannot be called . . . 33
6.2.3 Avoid deleting the subquery data once it reaches the final state . . . 34
6.2.4 `SubQueryEngine.subqueryStatus` does not validate the query-subquery relationship . . . 35
6.2.5 Improve the sanity checks of the `QueueStrategyEngine.setQueues` . . . 35
6.2.6 `QueueStrategyEngine.allocate` could allocate more than the cap target . . . 36
6.2.7 `QueueStrategyEngine` functions `allocate` and `unallocate` should ”simulate” the outcome to avoid reverts . . . 38
6.2.8 `QueueStrategyEngine.unallocate` should not allow unallocations for vehicle's configured with ”manual” operating mode . . . 38
6.2.9 `QueueStrategyEngine.unallocate` could unallocate more than the cap target upper bound 39
6.2.10 Additional validations across the `QueryRedeemQueue` contract's logic . . . 40
6.2.11 Partial redemptions uses incorrect rounding in `totalSharesFulfilled` calculations . . . 40
6.2.12 `MultiVehicle.exitSupplies` can return outdated values . . . 41
6.2.13 The `MultiVehicleFacets` threshold `extraAssetsForWithdrawalRequests` should be upper bounded . . . 42
6.2.14 Consider refactoring the `maxDepositable` and `maxRedeemable` functions in `SectorAccountingEngine` to exclude donations . . . 43
6.2.15 `_dispatchDeposit` and `_dispatchRedeem` should revert when `DispatchParams` is not empty and returned state is `EMPTY` . . . 44
6.2.16 Revert due to zero amount fulfillment may break operations from the MultiVehicle . . . 45
6.2.17 `subQueryStatus` should return `EXEC` for `PAUSED` queries . . . 49
6.2.18 The `rebalance` function behavior must be refactored . . . 50
6.2.19 ”Manual” dispatches could ignore the `minOutput` expressed in the `DispatchParams` requirement . . . 50
6.2.20 `lookup()` can loop indefinitely when no fulfillment matches . . . 52

6.2.21 `QueueStrategyEngine.allocate` should not use the same `Target` to both validate the max exposure and delta target . . . 53
6.2.22 `_dispatchDeposit()` and `_dispatchRedeem()` Early Returns Leave VEHICLE Sector Assets Stranded . . . 54
6.2.23 User slippage protection is ineffective on MultiVehicle redeem queries . . . 55
6.2.24 `unallocate()` `complete` Return Value Ignored Causing Silent Liveness Failure . . . 56
6.2.25 `_createRedeem` could return less assets than the expected amount . . . 57
6.2.26 Functions that could change total assets/shares must be routed via `MultiVehicle` and wrapped with Fee Manager fee logic . . . 58
6.2.27 Vehicle exposure caps can be underestimated because `sharesBeforeCreates` are excluded 59
6.2.28 Stale `totalAssets_` in `_createRedeem()` After `requestWithdrawable` . . . 60
6.2.29 Unauthorizing an async vehicle with a PROCESSING query strands its sector funds . . . 61
6.2.30 Residual `maxAmountOut` should be reduced proportionally, not by the amount paid . . . 61
6.3 Informational . . . 62
6.3.1 Dead code . . . 62
6.3.2 Ineffective logic in `constrainBy` . . . 63
6.3.3 Bulk Informational Issues . . . 64
6.3.4 Natspec issues . . . 67
6.3.5 Cross-contract sanity checks during MultiVehicle initialization . . . 68
6.3.6 `VehicleRegistry` should be refactored . . . 69
6.3.7 Considerations and side effects relative to the assumptions for the `ready()` vehicle's behavior 70
6.3.8 Document unclear or unspecified implicit assumptions . . . 71
6.3.9 Consider refactoring part of the `QueueStrategyEngine.allocate` code . . . 72
6.3.10 `_createDeposit` can not compute non-zero extra shares and non-zero missing shares . . . 72
6.3.11 `QueryRedeemQueue.redeem` could return the pending amount . . . 73
6.3.12 Consider tracking with an event the amount not allocated/unallocated during the `deposit` and `requestWithdrawable` execution . . . 74
6.3.13 Consider refactoring the `SectorAccountingEngine.syncVehicleActivationStatus` function for a better and more secure DX . . . 74
6.3.14 `_totalUnaccountQuery` loop refactor . . . 75
6.3.15 `VehicleRegistry` threshold configuration sanity checks . . . 76
6.3.16 Consider refactoring the `SectorAccountingEngine._isActiveVehicle` function . . . 77
6.3.17 Consider returning and ”bubbling up” the result of the internal `_dispatch*` operations in the `_dispatch` function . . . 78
6.3.18 `_vehicleHoldings` Comments Do Not Document Recovery Scenario for Ephemeral Accounting . . . 79
6.3.19 STEAM and vehicle specifications . . . 79
6.3.20 `Target.remainingCapacity` should be used for capacity calculations . . . 80
6.3.21 Threshold tolerance should only be used during allocation calculation . . . 81
6.3.22 `BaseVehicle._payout()` Does Not Guard Against Zero-Amount Transfers . . . 81
6.3.23 `moveShares` action can have different outcomes . . . 82
6.3.24 Async Redeem Funds Received After Queue Fulfillment By Deposits Sit Idle in REDEEM Sector 82
6.3.25 Keeper assumptions . . . 83
6.3.26 `estimate()` security considerations . . . 83
6.3.27 `QueryRedeemQueue` security considerations . . . 85
6.3.28 MultiVehicle does not support partial success from queries in underlying vehicles . . . 85
6.3.29 `_isActiveVehicle` may treat a vehicle as inactive while it has on-going queries . . . 86
6.3.30 `SectorAccountEngine` uses zero minimum output as slippage parameter . . . 87
6.3.31 Pending redeem accounting can cause withdrawal liquidity to be overestimated . . . 88
6.3.32 `_dispatchDeposit` and `_dispatchRedeem` silent fail behavior are unclear and not documented . . . 89

7 Appendix **91**

7.1 Engagement structure 91
7.2 Focus areas 91
7.2.1 Open or partially resolved findings 91
7.2.2 Fix-review follow-ups 92
7.2.3 General caution 92
7.3 End-to-end testing: configurations exercised 93
7.4 Automatic allocations to vehicles with deposit fees break share fairness: Proof of Concept 93

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Kiln: Phase 2 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 14 days in total, Kiln engaged with Spearbit to review the railnet protocol. In this period of time a total of **68** issues were found.

Summary

Project Name	Kiln
Repository	railnet
Commit	e3375859
Type of Project	DeFi, Vault
Audit Timeline	Apr 7th to Apr 21st

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	6	4	2
Low Risk	30	20	10
Gas Optimizations	0	0	0
Informational	32	29	3
Total	**68**	**53**	**15**

4.1 Scope

The security review had the following components in scope for railnet on commit hash e3375859:

```
├── factories
│   └── vehicles
│       └── MultiVehicleFactory.sol
├── jobs
│   └── MultiVehicleJobListing.sol
├── keeper
│   ├── interfaces
│   │   └── IReceiver.sol
│   └── Keeper.sol
└── vehicles
    └── multi
        ├── abstracts
        │   ├── MultiVehicleErrors.sol
        │   ├── MultiVehicleEvents.sol
        │   └── MultiVehicleSharedInternal.sol
        ├── facets
        │   └── MultiVehicleFacets.sol
        ├── interfaces
```

```
│   ├── IFeedQueryRedeemQueueFacet.sol
│   ├── IQueryRedeemQueue.sol
│   ├── IQueueStrategyEngine.sol
│   ├── IRetrieveQueryRedeemQueueAssetsFacet.sol
│   ├── ISectorAccountingEngine.sol
│   ├── ISetThresholdsFacet.sol
│   ├── ISubQueryEngine.sol
│   └── IVehicleRegistry.sol
├── libs
│   ├── MultiVehicleStore.sol
│   ├── MultiVehicleStructs.sol
│   ├── Sector.sol
│   ├── SubQuery.sol
│   └── Target.sol
├── MultiVehicle.sol
├── QueryRedeemQueue.sol
├── QueueStrategyEngine.sol
├── SectorAccountingEngine.sol
├── SubQueryEngine.sol
├── VEHICLE.md
└── VehicleRegistry.sol
```

5 Overview

Kiln's Phase 2 audit covers the **MultiVehicle**: a STEAM-compliant *meta-vehicle* whose underlying ”protocol” is a portfolio of other STEAM vehicles. Users deposit a single base asset and receive MultiVehicle shares backed by positions spread across authorized sub-vehicles.

A MultiVehicle deployment consists of six cooperating beacon-proxied contracts. The **MultiVehicle** itself is the user-facing vault; the **SectorAccountingEngine** is a double-entry ledger that tracks all assets and shares through logical *sectors* (deposit, allocation, redeem, plus per-vehicle and per-query staging sectors); the **QueueStrategyEngine** holds the allocation policy as priority-ordered deposit and redeem queues with per-vehicle targets; the **SubQueryEngine** creates and progresses queries against sub-vehicles and keeps *ephemeral accounting* — estimate-based bookkeeping that keeps `totalAssets()` accurate while sub-queries are in flight; the **QueryRedeemQueue** is a FIFO demand/fulfillment queue for redemptions that exceed available liquidity; and the **VehicleRegistry** manages which sub-vehicles may receive allocations, including detection of cyclical compositions.

Deposits settle synchronously and are then allocated down the deposit queue. Redemptions are served from available liquidity first, then by unallocating from sub-vehicles, and any remainder is queued as a demand that keepers later fulfill — so a redemption can settle immediately, partially, or fully asynchronously. An optional auto-redemption mechanism lets new deposits recycle queued redemption demands instead of minting fresh shares.

The asynchronous flows are automated by keeper infrastructure: the `MultiVehicleJobListing` advertises recurring jobs per MultiVehicle (feeding the redeem queue with available liquidity and retrieving excess queue assets), and the on-chain `Keeper` contract executes batched job reports submitted by an authorized off-chain forwarder. The `MultiVehicleFactory` deterministically deploys and wires the full set of components behind upgradeable beacons, seeding each new instance with a burned initial deposit.

6 Findings

6.1 Medium Risk

6.1.1 Inaccurate `isVehicleSector` masking allows to manipulate `totalAssets`

Severity: Medium Risk.

Context: Sector.sol#L103-L105.

Description: The `isVehicleSector` checks that a given sector is a vehicle sector by masking the first 12 bytes of the 32-byte sector.

```
/// @notice Mask constant for identifying sector types.
bytes12 private constant SECTOR_PREFIX_MASK = 0xFF0000000000000000000000;


function isVehicleSector(Sector self) internal pure returns (bool) {
    return bytes12(Sector.unwrap(self)) & SECTOR_PREFIX_MASK == VEHICLE_SECTOR_PREFIX;
}
```

This function logic checks `bytes12(sector) & 0xFF0000000000000000000000 == 0x010000000000000000000000` . As shows this mask, only the first byte is checked.

However, a vehicle sector is expected to be `bytes12(VEHICLE_SECTOR_PREFIX) | bytes20(vehicleAddress)` . With the current logic, a sector with `bytes[1:12]` that are non-zero would be recognized as a vehicle sector while it should not.

Impact: By exploiting the incorrect masking, the entity with the `MULTI_VEHICLE_MOVE_ASSETS` role can move assets to an unrecognized vehicle sector.

This leads to manipulating the `MultiVehicle.totalAssets` value as the assets moved to this unrecognized sector will not be accounted.

Recommendation: Check that the sector first 12 bytes correspond to the expected mask.

```
function isVehicleSector(Sector self) internal pure returns (bool) {
    return bytes12(Sector.unwrap(self)) == VEHICLE_SECTOR_PREFIX;
}
```

Kiln: Fixed in commit 0ca757ff. Fixed by applying the recommendation. `Sector.isVehicleSector` uses exact 12-byte equality and the old SECTOR_PREFIX_MASK constant has been removed.

Spearbit: Fixed. The recommendation has been applied. Regression tests were added.

6.1.2 Problems and side effects of the `QueryRedeemQueue.retrievable` behavior

Severity: Medium Risk.

Description: The `QueryRedeemQueue.retrievable` state variable tracks the amount of excess assets that has been accumulated during the fulfillment phase compared to the demand, typically due to revenue generation during the time between the redeem query creation and fulfillment execution. The current implementation of `MultiVehicle._totalAssets()` , which tracks the total assets owned by the `MultiVehicle` itself is including this excess of assets tracked by the `QueryRedeemQueue.retrievable` .

```
function _totalAssets() internal view override returns (uint256) {
    MultiVehicleStore.Storage storage $ = MultiVehicleStore.getStorage();
    return $.accountingEngine.totalAssets() + $.redeemQueue.retrievable();
}
```

The increase of the `retrievable` variable happens only when a user executes the `unlock` operation on a query that is bound to a `demandId`.

Performance fees lost for the Fee Managers: The performance fees of the Fee Manager are based on the difference between the current `totalAssets()` and the "cached" total assets that are stored in the `$.cache[MultiVehicle]` state variable (in the `FeeManager` contract) which updated at the end of each `BaseVehicle` operation by executing `_feeManager.onUpdate(totalAssets());`.

If the `MultiVehicleFacets.unlock` increases the `QueryRedeemQueue.retrievable`, the Fee Manager will lose the performance fees on that amount. This happens because the fees have already been calculated on the previous `totalAssets()` (calculated before the increase) and the new `totalAssets()` (which includes the increase) will override what is tracked by the `FeeManager` cache once the `BaseVehicle` executes `_feeManager.onUpdate(totalAssets())` (triggered **after** `MultiVehicleFacets.unlock`).

Sandwiching unlock operations for an extra instant profit: This behavior allows users to game the timing to enter and exit the market based on `unlock` operations that can "spike" the `totalAssets` with "donations" that will indeed increase the share price. Users with this knowledge can simply "sandwich" those unlock operations to gain an instant profit and "eat" part of that surplus that was entitled to pre-existing holders.

Recommendation: Kiln has two options:

1) Refactor **when** the `retrievable` is calculated and accounted for. The correct time to increase `retrievable` is during the `fulfill` execution.
2) Remove the `QueryRedeemQueue.retrievable` from the `_totalAssets` calculation and only allow authed users (via role) to withdraw it from the `QueryRedeemQueue` balance. This extra yield can be later on redistributed to existing suppliers in a proper way.

Kiln: Partially fixed in PR 448 (`retrievable` removed from `totalAssets`).

Partially fixed by applying recommendation #2: `retrievable` has been removed from `MultiVehicle._totalAssets()`. We confirm both residual impacts and agree the PR 448 changes relocate rather than fully eliminate the issue:

- Performance fees on the surplus are not captured. `VehicleManager.retrieveQueryRedeemQueueAssets` performs the reinjection inside the `startOngoingFeeHandling` / `finishOngoingFeeHandling` window: fee shares are minted against the pre-injection `totalAssets()`, and the post-injection `totalAssets()` becomes the new FeeManager checkpoint via `onUpdate`. The retrieved amount therefore never appears in any earnings window and accrues no performance fee.
- The reinjection remains sandwichable. `QueryRedeemQueue.retrievable()` is publicly observable and the reinjection is a step increase in share price, so a depositor entering before and exiting after the retrieve call can capture a pro-rata share of the surplus.

We accept this as a residual Medium for the following reasons:

- The surplus is bounded by yield accrued between redeem query creation and fulfillment, on the redeemed portion only: small, slowly-accumulating amounts relative to TVL.

- The reinjection endpoint is role-gated (`MULTI_VEHICLE_RETRIEVE_QUERY_REDEEM_QUEUE_ASSETS`). Operators control the amount and timing, can split retrievals into smaller tranches, and will submit retrieval transactions through private bundles, removing the front-running vector. Where the keeper job (`MultiVehicleJobListing.retrieveQueryRedeemQueueAssets`) is used, keeper submissions will follow the same private-submission policy, since the threshold-based trigger would otherwise make retrieval timing predictable.
- The structural hardening from PR 448 is preserved: the surplus is excluded from `totalAssets()` while it sits in the queue, and no unprivileged user can trigger the donation (the previous auto-injection during unlock is removed).

We may revisit the distribution mechanism in a future release (e.g., linear vesting of the retrieved surplus into `totalAssets()` , or an external reward distribution), which would address both legs structurally. For this engagement, the finding stands as Acknowledged.

Spearbit: Acknowledged. The PR 448 changes relocate rather than eliminate the issue: retrieval still happens inside the `startOngoingFeeHandling` / `finishOngoingFeeHandling` flow, so performance fees on the surplus are still lost, and the surplus remains observable on-chain, so the retrieve call can still be sandwiched. Kiln accepts this as a residual Medium and may revisit the distribution mechanism in a future release; the item is listed in the appendix (see the "Focus point document").

6.1.3 MultiVehicle's `totalAssets()` ignores sub-vehicle redeem fees, inflating share price and fees

Severity: Medium Risk.

Context: SectorAccountingEngine.sol#L632.

Summary: `SectorAccountingEngine.totalAssets()` over-values the MultiVehicle's holdings. At SectorAccountingEngine.sol:658 it calls `_vehicle.convertSingleAssetToAssets(_shares)` , which routes to `vehicle.convert(_assets, true)` (Vehicle.sol:80) with `ignoreTransactionalFees = true` . Sub-vehicle shares are valued at gross NAV, but realizing them requires paying each sub-vehicle's redeem fee.

The error grows with the number of fee-charging sub-vehicles and compounds across nested MultiVehicle layers.

Impact:

- **First-mover redeem extracts more than fair share.** When the DEPOSIT buffer covers a redeem, no sub-vehicle exit fee is paid but the redeemer is priced at gross NAV. Late redeemers eat the realized fees which means it incentivizes bank-run.
- `maxAmountOut` **overestimates**.
- **Incorrect fee estimations.** Performance and management fees are calculated based on an inflated `totalSupply` which will be incorrect.

Proof of Concept: Initial state: Alice and Bob each hold 100 MV shares (`totalSupply = 200`); DEPOSIT sector holds 100 base assets; 100 base assets are allocated to a single sub-vehicle with **50% redeem fee** (so its gross NAV in the MV = 100).

Reported `totalAssets()` = 100 + 100 = **200**; realizable value = 100 + 100 x 0.5 = **150**.

1. Alice redeems 100 shares. `_createRedeem` computes `100 · 200 / 200 = 100` . Buffer covers it; no `unallocate` runs. Alice receives **100**.
2. Bob redeems 100 shares. Buffer empty → `unallocate` redeems the allocation. Sub-vehicle pays 100 x 0.5 = 50. Bob receives **50**.

Identical positions, asymmetric payouts; the 50-asset gap is the exit fee Alice silently shifted to Bob.

Scenario above explains the `The` test_topMultivehicle_first_exit_drains_liquid_assets_and_second_exit_is_-fee_constrained' test shown below:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity >=0.8.33;

import {IVehicle} from "src/steam/IVehicle.sol";
import {Roles} from "src/libs/Roles.sol";
import {Target} from "src/vehicles/multi/libs/Target.sol";
import {IBaseVehicle} from "src/vehicles/base/interfaces/IBaseVehicle.sol";
import {FeeManager, IFeeManager} from "src/vehicles/base/FeeManager.sol";
import {FeeManagerFactory} from "src/factories/vehicles/FeeManagerFactory.sol";
import {ModulesManager} from "src/vehicles/base/ModulesManager.sol";
import {MultiVehicle} from "src/vehicles/multi/MultiVehicle.sol";
import {QueueStrategyEngine} from "src/vehicles/multi/QueueStrategyEngine.sol";
import {SectorAccountingEngine} from "src/vehicles/multi/SectorAccountingEngine.sol";
import {VehicleRegistry} from "src/vehicles/multi/VehicleRegistry.sol";
import {IQueueStrategyEngine} from "src/vehicles/multi/interfaces/IQueueStrategyEngine.sol";
import {SectorLib} from "src/vehicles/multi/libs/Sector.sol";
import {MultiVehicleFactory} from "src/factories/vehicles/MultiVehicleFactory.sol";
import {Interceptor} from "src/abstracts/Interceptor.sol";
import {Query, State} from "src/steam/Query.sol";

import {MultiVehicleTestInternals} from "test/vehicles/multi/MultiVehicleTestInternals.sol";

/// @title Nested MultiVehicle Redeem Fee NAV Standalone Test
/// @notice Shows that top-level MultiVehicle NAV can overstate what a full unwind actually
↪  realizes
///         when nested child MultiVehicles charge redeem payout fees.
contract MultiVehicleNestedRedeemFeeNAVStandaloneTest is MultiVehicleTestInternals {
    using SectorLib for IBaseVehicle;

    struct NestedMV {
        MultiVehicle mv;
        SectorAccountingEngine ae;
        QueueStrategyEngine se;
        VehicleRegistry vr;
    }

    uint32 internal constant HIGH_REDEEM_FEE_BPS = 5_000; // 50%

    function test_topMultivehicle_totalAssets_overstates_full_unwind_value_under_nested_redeem
    ↪  _fees() external {
        _grant(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION, address(this));
        _grant(Roles.MULTI_VEHICLE_SET_QUEUES, address(this));

        FeeManager _bottomFeeManager =
        ↪  _deployFeeManagerWithRedeemFee(keccak256("BOTTOM_FEE_MANAGER"));
        FeeManager _middleFeeManager =
        ↪  _deployFeeManagerWithRedeemFee(keccak256("MIDDLE_FEE_MANAGER"));

        NestedMV memory _bottom = _deployNestedMultiVehicle(keccak256("BOTTOM_MULTI_VEHICLE"),
        ↪  _bottomFeeManager);
        NestedMV memory _middle = _deployNestedMultiVehicle(keccak256("MIDDLE_MULTI_VEHICLE"),
        ↪  _middleFeeManager);

        _authorizeVehicle(_middle.vr, IBaseVehicle(address(_bottom.mv)));
        _setSingleVehicleQueues(_middle.se, IBaseVehicle(address(_bottom.mv)));

        _authorize_vehicle(IBaseVehicle(address(_middle.mv)));
```

```
    _single_vehicle_queues(IBaseVehicle(address(_middle.mv)));

    address _user = makeAddr("user");
    _quick_deposit(_user, _unit(100));

    // Deposit chain is fully synchronous:
    //   Top seed 1 + user 100 = 101
    //   Middle seed 1 + Top deposit 101 = 102
    //   Bottom seed 1 + Middle deposit 102 = 103
    uint256 _grossTopAssets = $multiVehicle.totalAssets();
    assertEq(_grossTopAssets, _unit(101), "top gross NAV mismatch");
    assertEq(_middle.mv.totalAssets(), _unit(102), "middle gross NAV mismatch");
    assertEq(_bottom.mv.totalAssets(), _unit(103), "bottom gross NAV mismatch");

    // First force the middle MV to unwind its bottom-MV position.
    // Bottom charges a 50% redeem fee, so middle's realizable assets are halved.
    uint256 _middleGrossAssets = _middle.mv.totalAssets();
    vm.prank(address(_middle.mv));
    _middle.ae.requestWithdrawable(_middleGrossAssets);

    uint256 _topAssetsAfterBottomUnwind = $multiVehicle.totalAssets();
    assertEq(_middle.mv.totalAssets(), _unit(51), "middle should only realize half after
    ↪  bottom redeem fee");
    assertEq(_topAssetsAfterBottomUnwind, _grossTopAssets / 2, "top view should reflect
    ↪  bottom fee loss");

    // Then force the top MV to unwind its middle-MV position.
    // Middle also charges a 50% redeem fee, so top realizes only half again.
    vm.prank(address($multiVehicle));
    $accountingEngine.requestWithdrawable(_topAssetsAfterBottomUnwind);

    uint256 _realizedTopWithdrawable = $accountingEngine.withdrawable();
    assertEq(_realizedTopWithdrawable, _grossTopAssets / 4, "top full unwind value
    ↪  mismatch");
    assertEq($multiVehicle.totalAssets(), _realizedTopWithdrawable, "top NAV should equal
    ↪  realized withdrawable");

    assertLt(_realizedTopWithdrawable, _grossTopAssets, "top gross NAV must overstate
    ↪  realizable value");
    assertEq(_topAssetsAfterBottomUnwind, _grossTopAssets / 2, "bottom fee should halve top
    ↪  view");
    assertEq(_realizedTopWithdrawable, _grossTopAssets / 4, "nested fees should quarter
    ↪  final unwind value");
}

function test_topMultivehicle_first_exit_drains_liquid_assets_and_second_exit_is_fee_const
↪  rained() external {
    _grant(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION, address(this));
    _grant(Roles.MULTI_VEHICLE_SET_QUEUES, address(this));
    _grant(Roles.MULTI_VEHICLE_MOVE_ASSETS, address(this));
    _grant(Roles.MULTI_VEHICLE_DISPATCH, address(this));

    FeeManager _childFeeManager =
    ↪  _deployFeeManagerWithRedeemFee(keccak256("CHILD_FEE_MANAGER"));
    NestedMV memory _child = _deployNestedMultiVehicle(keccak256("CHILD_MULTI_VEHICLE"),
    ↪  _childFeeManager);
    IBaseVehicle _childVehicle = IBaseVehicle(address(_child.mv));

    _authorize_vehicle(_childVehicle);

    IQueueStrategyEngine.QueueEntry[] memory _depositQueue = new
    ↪  IQueueStrategyEngine.QueueEntry[](0);
```

```
IQueueStrategyEngine.QueueEntry[] memory _redeemQueue = new
↪  IQueueStrategyEngine.QueueEntry[](1);
_redeemQueue[0] = IQueueStrategyEngine.QueueEntry({
    vehicle: _childVehicle, target: Target({value: 0, threshold: 0})
});
_set_queues(_depositQueue, _redeemQueue);

address _alice = makeAddr("alice");
address _bob = makeAddr("bob");

_quick_deposit(_alice, _unit(100));
_quick_deposit(_bob, _unit(100));

assertEq($multiVehicle.balanceOf(_alice), _unit(100), "alice shares mismatch");
assertEq($multiVehicle.balanceOf(_bob), _unit(100), "bob shares mismatch");

// The top-level MV is seeded with 1 burned share. Move that 1 asset plus another 100
↪  assets
// into the child so the top MV keeps exactly 100 liquid assets for the first redeemer.
uint256 _childPositionAssets = _unit(101);
uint256 _expectedBobImmediateAssets = (_childPositionAssets * (10_000 -
↪  HIGH_REDEEM_FEE_BPS)) / 10_000;
uint256 _expectedBobQueuedShares = _unit(100) - _expectedBobImmediateAssets;

$accountingEngine.moveAssets(SectorLib.DEPOSIT, _childVehicle.toSector(),
↪  _childPositionAssets);
_dispatch(_childVehicle);

assertEq(_sectorBalance(SectorLib.DEPOSIT, $asset), _unit(100), "deposit sector should
↪  keep 100 liquid assets");
assertEq(
    _sectorBalance(SectorLib.ALLOCATION, _childVehicle),
    _unit(101),
    "child allocation should hold the remaining NAV"
);
assertEq($multiVehicle.totalAssets(), _unit(201), "top NAV mismatch after staging");

Query memory _aliceRedeem = _quick_redeem(_alice, _unit(100), State.UNLOCKING);
assertEq(uint256($multiVehicle.state(_aliceRedeem)), uint256(State.UNLOCKING), "alice
↪  redeem should unlock");
assertEq($asset.balanceOf(_alice), 0, "alice should not receive assets before unlock");

vm.prank(_alice);
$multiVehicle.unlock(_aliceRedeem);

assertEq($asset.balanceOf(_alice), _unit(100), "alice should drain the liquid deposit
↪  assets");
assertEq(uint256($multiVehicle.state(_aliceRedeem)), uint256(State.SETTLED), "alice
↪  redeem should settle");
assertEq(_sectorBalance(SectorLib.DEPOSIT, $asset), 0, "first exit should empty the
↪  deposit sector");
assertEq($multiVehicle.totalAssets(), _unit(101), "remaining NAV should sit in the
↪  child position");

Query memory _bobRedeem = _quick_redeem(_bob, _unit(100), State.UNLOCKING);
uint256 _bobDemandId = $multiVehicle.demand(_bobRedeem);

assertGt(_bobDemandId, 0, "bob should leave queued demand behind");
assertEq(uint256($multiVehicle.state(_bobRedeem)), uint256(State.UNLOCKING), "bob
↪  redeem should unlock");
assertEq($asset.balanceOf(_bob), 0, "bob should not receive assets before unlock");
```

```
        vm.prank(_bob);
        $multiVehicle.unlock(_bobRedeem);

        assertEq(
            $asset.balanceOf(_bob),
            _expectedBobImmediateAssets,
            "bob should only receive the fee-constrained child redeem"
        );
        assertEq(
            uint256($multiVehicle.state(_bobRedeem)),
            uint256(State.PROCESSING),
            "bob redeem should remain queued after the partial unlock"
        );
        assertEq(
            $redeemQueue.pending(_bobDemandId), _expectedBobQueuedShares, "bob should still
            ↪  have the remaining redeem queued"
        );
        assertEq($redeemQueue.unredeemable(), _expectedBobQueuedShares, "the queue should
        ↪  retain the remaining half");
    }

    function _deployFeeManagerWithRedeemFee(bytes32 salt) internal returns (FeeManager
    ↪  feeManager) {
        IFeeManager.FeeRecipient[] memory _recipients = new IFeeManager.FeeRecipient[](1);
        _recipients[0] = IFeeManager.FeeRecipient({target: makeAddr("fee_recipient"),
        ↪  shareBps: 10_000});

        (feeManager,) = _deployFeeManager(
            $coreFactory,
            FeeManagerFactory.SpawnParams({
                accessControl: $accessControl,
                initialFees: IFeeManager.Fees({
                    performanceFeeBps: 0, managementFeeBps: 0, depositFeeBps: 0, redeemFeeBps:
                    ↪  HIGH_REDEEM_FEE_BPS
                }),
                initialMaxFees: IFeeManager.Fees({
                    performanceFeeBps: 0, managementFeeBps: 0, depositFeeBps: 0, redeemFeeBps:
                    ↪  HIGH_REDEEM_FEE_BPS
                }),
                initialRecipients: _recipients,
                deploymentSalt: salt
            })
        );
    }

    function _deployNestedMultiVehicle(bytes32 salt, FeeManager feeManager) internal returns
    ↪  (NestedMV memory nested) {
        MultiVehicleFactory.Salts memory _salts = MultiVehicleFactory.Salts({
            multiVehicle: salt,
            queryRedeemQueue: keccak256(abi.encodePacked(salt, "QRQ")),
            queueStrategyEngine: keccak256(abi.encodePacked(salt, "QSE")),
            sectorAccountingEngine: keccak256(abi.encodePacked(salt, "SAE")),
            subQueryEngine: keccak256(abi.encodePacked(salt, "SQE")),
            vehicleRegistry: keccak256(abi.encodePacked(salt, "VR")),
            initialDepositQuery: keccak256(abi.encodePacked(salt, "IDQ"))
        });

        (MultiVehicleFactory.Contracts memory _contracts,) = _deployMultiVehicle(
            $coreFactory,
            MultiVehicleFactory.SpawnParams({
                asset: $asset,
                accessControl: $accessControl,
```

```
                feeManager: feeManager,
                modulesManager: ModulesManager(address(0)),
                initialInterceptions: new Interceptor.Interception[](0),
                salts: _salts,
                name: "Nested",
                symbol: "NESTED",
                forbiddenAddresses: new address[](0)
            })
        );

        nested.mv = MultiVehicle(_contracts.multiVehicle);
        nested.ae = SectorAccountingEngine(_contracts.sectorAccountingEngine);
        nested.se = QueueStrategyEngine(_contracts.queueStrategyEngine);
        nested.vr = VehicleRegistry(_contracts.vehicleRegistry);
    }

    function _authorizeVehicle(VehicleRegistry registry, IBaseVehicle vehicle) internal {
        registry.authorize(IVehicle(address(vehicle)));
    }

    function _setSingleVehicleQueues(QueueStrategyEngine strategyEngine, IBaseVehicle vehicle)
    ↪   internal {
        IQueueStrategyEngine.QueueEntry[] memory _depositQueue = new
        ↪   IQueueStrategyEngine.QueueEntry[](1);
        IQueueStrategyEngine.QueueEntry[] memory _redeemQueue = new
        ↪   IQueueStrategyEngine.QueueEntry[](1);

        _depositQueue[0] = IQueueStrategyEngine.QueueEntry({
            vehicle: vehicle, target: Target({value: type(uint256).max, threshold: 0})
        });
        _redeemQueue[0] = IQueueStrategyEngine.QueueEntry({vehicle: vehicle, target:
        ↪   Target({value: 0, threshold: 0})});

        strategyEngine.setQueues(_depositQueue, _redeemQueue);
    }
}
```

Recommendation: Subtract the sub-vehicle exit fee when valuing sub-vehicle holdings for `totalAssets()`. At SectorAccountingEngine.sol:658, replace `convertSingleAssetToAssets` with a fee-aware helper that calls `vehicle.estimate(shares, REDEEM, OUTPUT)` (or `vehicle.convert(_assets, false)`) so the returned value is net of the redeem fee the MultiVehicle would actually pay.

Kiln: Fixed in commit 77d03ca7.

Fixed by applying the recommendation — `SAE.totalAssets()` now values sub-vehicle shares via `estimateSingleAssetAssets(_, true)` (redeem-output estimate, post-fee) instead of the face-value `convertSingleAssetToAssets`. `ERC4626Vehicle._totalAssets` and `EthenaVehicle._totalAssets` switched to `previewRedeem(totalUnderlyingShares)` for parity. With the delta-based mint in the same commit, redeem-fee is pre-charged at deposit time and `previewRedeem` / actual `withdraw()` payout agree.

Spearbit: The fix has been applied. The review of the new fee-aware `totalAssets()` was not completed during the engagement; it warrants particular attention in the next audit and is listed in the appendix (see the "Focus point document"). Specifically, confirm that using the post-fee redeem estimate for sub-vehicle shares in `totalAssets()` does not cause over-minting in the DEPOSIT flow, i.e. that the delta-based mint correctly accounts for the lower, fee-adjusted NAV. One option is to use different variants of `totalAssets` for the deposit and redeem paths, depending on where each is consumed.

6.1.4 Queued redeem demand inflates child MV NAV, Parent MV deposits at understated share price

Severity: Medium Risk.

Context: QueryRedeemQueue.sol#L240-L258.

Summary: When a redeem demand is queued in the child MultiVehicle's `QueryRedeemQueue`, the queued shares stay in `totalSupply` and the assets backing them remain in the child's accounting until the demand is fulfilled. The demand's payout is fixed at `maxAmountOut` at queue time, so any post-queue yield on those assets economically belongs to the remaining holders, not to the queued claim. The child MV's `convert` / `totalAssets` / `totalSupply` math does not net out the queued claim, so the per-share price it reports stays at `(totalAssets) / (totalSupply)` instead of the fair `(totalAssets - sum(queued maxAmountOut)) / (totalSupply - queued shares)`.

A Parent MV holding child shares values its position via the child's reported NAV (SectorAccountingEngine.sol:658), so the Parent's own `totalAssets` is *understated* during the queued window, and depositors into the Parent mint at an unfairly low share price.

Impact:

- **Parent MV depositors are over-minted.** While a queue claim is outstanding in the child, the Parent's per-share NAV is below fair value. New depositors get more Parent shares per asset than they should. After the child queue resolves and the inflated yield is redistributed across the *non-queued* child shares, the over-minted Parent shares appreciate, letting the attacker redeem at a profit.
- **Existing Parent holders are diluted.** The over-minting comes out of their share value.

Proof of Concept: State:

- Child MV: 100 shares, 100 USDC assets, NAV = 1.0/share.
- Parent MV holds 60 child shares; Alice holds 40 child shares.

1. Alice queues a redeem of 40 child shares at NAV = 1.0. Her queued claim is fixed at `maxAmountOut = 40 USDC`.
2. Child's underlying yields, total assets grow 100 → 120 USDC. Alice's claim remains 40 (fixed at queue time); the extra 20 economically belongs to the remaining 60 live shares, so the fair per-share price for live shares is `(120 - 40) / 60 = 80 / 60 ≈ 1.333`.
3. Child's reported NAV is still `120 / 100 = 1.2/share` (queued shares not netted out).
4. Parent MV values its 60 child shares at `60 · 1.2 = 72 USDC` instead of the fair `60 · 1.333 = 80 USDC`. Parent's `totalAssets` is understated by 8 USDC.
5. Attacker deposits 72 USDC into Parent MV. With Parent's understated NAV, they mint shares as if buying 50% of the pool (72 / (72 + 72) by mint math), instead of the fair 72 / (80 + 72) ≈ 47.4%.
6. Alice's queued redeem fulfills, drains 40 USDC from child to Alice. Child now has 80 USDC / 60 shares = 1.333/share. Parent's 60 child shares are now correctly valued at 80 USDC. Parent's `totalAssets` rises from the (incorrect) 72 to the (correct) 80 + attacker's 72 = 152.
7. Attacker redeems their over-minted Parent shares at the corrected NAV and walks away with more than the 72 they deposited.

The attacker's profit is the excess minting they captured, paid out of existing Parent holders' share value.

Recommendation: When computing per-share value, the MultiVehicle should account for outstanding queued redeem obligations so that yield earned after a demand is queued is not mispriced into shares that no longer have a claim on it. Any path that derives share price, NAV, or upstream valuation from raw `totalAssets` / `totalSupply` should instead use values that reflect the assets and supply actually backing live, non-queued shareholders.

Kiln: Acknowledged — intentional design choice.

The yield that economically accrues to live (non-queued) holders during a queued window is not auto-netted from totalAssets(). The mechanism to re-inject it back to holders already exists: VehicleManager.retrieveQueryRedeemQueueAssets(amount) (role-gated, see the `QueryRedeemQueue.retrievable` finding and PR 448) pulls the retrievable surplus from QueryRedeemQueue into the MV where today it credits holders entirely. Auto-netting is deferred to keep open a future FeeManager-aware split on this specific surplus that is under consideration — encoding a netting rule now would foreclose that design space. The asset manager decides the schedule and proportion of each re-injection on their own terms. Nested MultiVehicle deployments inherit this: parent NAV is correct at each retrieve boundary but may transiently understate during a queued window; asset managers running nested setups must coordinate their retrieve schedules accordingly.

Spearbit: Acknowledged. This behaviour should be clearly documented: the parent's share calculation will be incorrect while the child's redeem queue is not empty, which matters for multi-layered vehicle structures. The item is listed in the appendix (see the "Focus point document") as requiring further verification: confirm whether the child MV's reported NAV correctly nets out queued-but-unfulfilled redeem demands, and that a parent MV's `totalAssets` is not understated during the queue window.

6.1.5 `requestWithdrawable` can modify `totalAssets` and let fulfillment have a better rate

Severity: Medium Risk.

Context: MultiVehicleFacets.sol#L348-L361.

Description: The `requestWithdrawable` call in `_createRedeem` may affect the `totalAssets` value due to redeem fees being taken by underlying vehicles (due to dispatched redeems being executed).

This could lead to an outdated `totalAssets_` value passed to `__tryAutoRedemption`. As `totalAssets_` would be overvalued, it would harm share holders and the demands in the query redeem queue will have a better rate.

```
function _createRedeem(Query calldata, Id qid, uint256 sharesToRedeem, uint256 totalSupply_,
↪  uint256 totalAssets_)
    internal
    returns (State, State[] memory)
{
    // ...

    // [3] Calculate total assets needed (current query + queued demands + extra buffer)
    // If insufficient, request unallocation from sub-vehicles via QueueStrategyEngine
    {
        (bool _success, uint256 _totalAssetsToRedeem) = Math.tryAdd(
            _assetsToWithdraw + _queryRedeemQueueAssetsToRedeem,
            ↪  $.thresholds.extraAssetsForWithdrawalRequests
        );
        if (!_success) {
            _totalAssetsToRedeem = type(uint256).max;
        }

        if (_withdrawable < _totalAssetsToRedeem) {
            // Request accounting engine to make assets withdrawable
            // This triggers unallocate() on QueueStrategyEngine, which returns redeem
            ↪  instructions
            // SubQueryEngine then creates redeem queries to sub-vehicles
```

```
            $.accountingEngine.requestWithdrawable(_totalAssetsToRedeem); // @audit may update
            ↪  total assets

            // [4] Re-check withdrawable after requestWithdrawable (may have increased if sync
            ↪  redeems)
            _withdrawable = $.accountingEngine.withdrawable();
        }
    }

    // [5] If there are queued demands, fulfill them first (FIFO fairness)
    // Use available withdrawable to feed the queue before fulfilling current redemption
    if (_queryRedeemQueueAssetsToRedeem > 0) {
        (, uint256 _totalSharesFulfilled) = __tryAutoRedemption(
            0, // extraAssets
            totalSupply_,
            totalAssets_, // @audit outdated total assets
            true // skipThresholdCheck (explicit call path)
        );

        // Burn shares funded by withdrawable assets (helper does not burn)
        // Since extraAssets=0, all shares came from withdrawable
        if (_totalSharesFulfilled > 0) {
            _burn(address(this), _totalSharesFulfilled);
            // Re-check withdrawable after fulfilling queue
            _withdrawable = $.accountingEngine.withdrawable();
        }
    }

    // ...
}
```

Impact: Underlying vehicles with redeem fees will break the share fairness. The redeem query triggering `requestWithdrawable` can end up receiving more assets than expected.

Proof of Concept: The proof of concept attached shows the following output.

```
[PASS] test_PoC_requestWithdrawableUsesStaleTotalAssetsForQueuedRedeems() (gas: 4993555)
Logs:
  Forking at block number from env: 25071042
  AAVE initial deposit amount:  1000000
  Morpho initial deposit amount:  1000000
  Initial deposit amount:  1000000
  Queued redeemer USDC:   9999999999
  Trigger redeemer USDC:  8001800000
  Aave redeem fee USDC:   2000199999
  MV Aave shares:         0
```

The proof of concept highlights that the ”queued redeemer” gets way more USDC than the ”trigger redeemer”.

```
// SPDX-License-Identifier: BUSL-1.1
//
//
//
//
//
//
//
//
```

```
//     ==================================================
//
pragma solidity >=0.8.33;

// solhint-disable kiln-rules/ordering
// solhint-disable no-console

import {Test, console} from "@forge-std/Test.sol";
import {IERC20} from "@openzeppelin-contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from
↪  "@openzeppelin-contracts/token/ERC20/extensions/IERC20Metadata.sol";
import {SafeERC20} from "@openzeppelin-contracts/token/ERC20/utils/SafeERC20.sol";
import {Math} from "@openzeppelin-contracts/utils/math/Math.sol";

import {IMorpho, Id as MarketId, MarketParams} from "@morpho/interfaces/IMorpho.sol";

// Railnet - common
import {Asset, EstimationType, Mode, Query, State} from "src/steam/Query.sol";
import {ExternalAccessControl} from "src/common/ExternalAccessControl.sol";
import {IExternalAccessControl} from "src/common/interfaces/IExternalAccessControl.sol";
import {FreezablePausableBeacon} from "src/common/FreezablePausableBeacon.sol";
import {SharesLib} from "src/libs/Shares.sol";
import {Roles} from "src/libs/Roles.sol";

// Railnet - factories
import {AccessControlFactory} from "src/factories/common/AccessControlFactory.sol";
import {AaveV3VehicleFactory} from "src/factories/vehicles/AaveV3VehicleFactory.sol";
import {ConduitFactory} from "src/factories/conduit/ConduitFactory.sol";
import {CoreFactory} from "src/factories/CoreFactory.sol";
import {FeeManagerFactory} from "src/factories/vehicles/FeeManagerFactory.sol";
import {MorphoBlueVehicleFactory} from "src/factories/vehicles/MorphoBlueVehicleFactory.sol";
import {MultiVehicleFactory} from "src/factories/vehicles/MultiVehicleFactory.sol";

// Railnet - vehicles
import {AaveV3Vehicle} from "src/vehicles/aave_v3/AaveV3Vehicle.sol";
import {IPool} from "src/vehicles/aave_v3/interfaces/IPool.sol";
import {IPoolAddressesProvider} from
↪  "src/vehicles/aave_v3/interfaces/IPoolAddressesProvider.sol";
import {TokenMath} from "@aave-v3-libs/helpers/TokenMath.sol";
import {Interceptor} from "src/abstracts/Interceptor.sol";
import {FeeManager} from "src/vehicles/base/FeeManager.sol";
import {ModulesManager} from "src/vehicles/base/ModulesManager.sol";
import {IBaseVehicle} from "src/vehicles/base/interfaces/IBaseVehicle.sol";
import {IFeeManager} from "src/vehicles/base/interfaces/IFeeManager.sol";
import {IVehicle} from "src/steam/IVehicle.sol";
import {MorphoBlueVehicle} from "src/vehicles/morpho_blue/MorphoBlueVehicle.sol";
import {MorphoBlueVehicleStructs} from
↪  "src/vehicles/morpho_blue/libs/MorphoBlueVehicleStructs.sol";
import {MultiVehicle} from "src/vehicles/multi/MultiVehicle.sol";
import {QueryRedeemQueue} from "src/vehicles/multi/QueryRedeemQueue.sol";
import {QueueStrategyEngine} from "src/vehicles/multi/QueueStrategyEngine.sol";
import {SectorAccountingEngine} from "src/vehicles/multi/SectorAccountingEngine.sol";
import {SubQueryEngine} from "src/vehicles/multi/SubQueryEngine.sol";
import {VehicleRegistry} from "src/vehicles/multi/VehicleRegistry.sol";
import {Target} from "src/vehicles/multi/libs/Target.sol";
import {IQueueStrategyEngine} from "src/vehicles/multi/interfaces/IQueueStrategyEngine.sol";

// Railnet - conduit
import {Conduit} from "src/conduit/Conduit.sol";
import {ConduitStructs} from "src/conduit/libs/ConduitStructs.sol";
import {IAccountList} from "src/conduit/interfaces/IAccountList.sol";
import {IOwnerRegistry} from "src/conduit/interfaces/IOwnerRegistry.sol";
```

```
// Test utils
import {Deployer} from "test/test_utils/Deployer.sol";
import {ForkUtils} from "test/test_utils/ForkUtils.sol";

/// @title Multilevel Railnet Fork Config
/// @notice Reusable scaffolding that wires three Railnet layers together against a real
↪  network:
///         - Level 0: AAVE V3 vehicle (USDC) and Morpho Blue vehicle (USDC market).
///         - Level 1: a MultiVehicle (USDC) that allocates between Aave and Morpho.
///         - Level 2: a Conduit linked to the MultiVehicle.
///         All Level 0 and Level 1 vehicles are configured with 10% performance fee, 1%
↪  management fee,
///         and 0% deposit / redeem fees.
contract MultilevelRailnetForkConfig is Test, Deployer {
    using SafeERC20 for IERC20;

    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */
    /*                                                                NETWORK
    ↪  */
    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */

    /// @dev Mainnet USDC.
    address internal constant $USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    /// @dev Aave V3 PoolAddressesProvider (mainnet).
    address internal constant $AAVE_POOL_ADDRESSES_PROVIDER =
    ↪  0x2f39d218133AFaB8F2B819B1066c7E434Ad94E9e;
    /// @dev Morpho Blue (deployed at the same address on every chain).
    address internal constant $MORPHO = 0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb;
    /// @dev Morpho Blue USDC / cbBTC market id (mainnet).
    MarketId internal constant $MORPHO_USDC_MARKET =
        MarketId.wrap(0x64d65c9a2d91c36d56fbc42d69e979335320169b3df63bf92789e2c8883fcc64);

    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */
    /*                                                          FEE PARAMETERS
    ↪  */
    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */

    /// @dev 10% performance fee.
    uint32 internal constant $PERFORMANCE_FEE_BPS = 1000;
    /// @dev 1% management fee.
    uint32 internal constant $MANAGEMENT_FEE_BPS = 100;
    /// @dev No deposit fee.
    uint32 internal constant $DEPOSIT_FEE_BPS = 0;
    uint32 internal constant $MAX_DEPOSIT_FEE_BPS = 1000;
    /// @dev No redeem fee.
    uint32 internal constant $REDEEM_FEE_BPS = 0;
    uint32 internal constant $MAX_REDEEM_FEE_BPS = 1000;

    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */
    /*                                                          DEPLOYED STATE
    ↪  */
    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */

    address public $deployer;
    address public $feeRecipient;
```

```
IERC20Metadata public $usdc;

CoreFactory public $coreFactoryRef;
ExternalAccessControl public $sharedAC;

// Level 0 - vehicles
AaveV3Vehicle public $aaveVehicle;
MorphoBlueVehicle public $morphoVehicle;
FeeManager public $aaveFeeManager;
FeeManager public $morphoFeeManager;

// Level 1 - multi vehicle
MultiVehicle public $multiVehicle;
QueueStrategyEngine public $strategyEngine;
SectorAccountingEngine public $accountingEngine;
QueryRedeemQueue public $redeemQueue;
SubQueryEngine public $subQueryEngine;
VehicleRegistry public $vehicleRegistry;
FeeManager public $multiFeeManager;

// Level 2 - conduit
Conduit public $conduit;
ConduitFactory public $conduitFactory;
FreezablePausableBeacon public $conduitBeacon;

/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */
/*                                                                  DEPLOY
↪  */
/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */

/// @notice Deploys every level. Must be invoked from a `setUp()` after the fork is
↪  selected.
function _deployAll() internal {
    $deployer = makeAddr("multilevel-deployer");
    $feeRecipient = makeAddr("multilevel-fee-recipient");
    $usdc = IERC20Metadata($USDC);

    $coreFactoryRef = _deployCoreFactory();
    $sharedAC = _deployAccessControlIfNeeded(
        $coreFactoryRef,
        AccessControlFactory.SpawnParams({
            initialDelay: 0,
            initialDefaultAdmin: address(this),
            initialRoles: new IExternalAccessControl.RoleAttribution[](0),
            deploymentSalt: keccak256("multilevel-shared-ac")
        })
    );

    // Make STEAM deposit/redeem permissionless so the SubQueryEngine can interact with
    ↪  sub-vehicles
    // and the Conduit can interact with the MultiVehicle. This mirrors the public role
    ↪  setup used
    // by `MultiVehicleTester` and avoids granting the role per address.
    $sharedAC.setRolePublic(Roles.VEHICLE_STEAM_DEPOSIT, true);
    $sharedAC.setRolePublic(Roles.VEHICLE_STEAM_REDEEM, true);

    // Register USDC in the asset registry with the canonical 1 USDC initial deposit
    ↪  amount.
    _registerAsset(_getOrDeployAssetRegistry(), $USDC);
```

```
    _deployLevel0();
    _deployLevel1();
    _deployLevel2();
}

/* ---------------------------------------------------------------------------------------------
↪  --------------------------------- */
/*                                                                  LEVEL 0
↪  */
/* ---------------------------------------------------------------------------------------------
↪  --------------------------------- */

function _deployLevel0() internal {
    // Per-vehicle fee managers - each gets the same 10% perf / 1% mgmt / 0/0 transactional
    ↪  split.
    $aaveFeeManager = _spawnFeeManager(keccak256("multilevel-fee-manager-aave"));
    $morphoFeeManager = _spawnFeeManager(keccak256("multilevel-fee-manager-morpho"));

    // AAVE V3 vehicle (USDC).
    AaveV3VehicleInfra memory _aaveInfra = _deployAaveV3VehicleInfra($deployer,
    ↪  $coreFactoryRef);
    uint256 _aaveExpectedSupply = _previewAaveInitialSupply($USDC);
    $aaveVehicle = _deployAaveV3VehicleInstance(
        $deployer,
        AaveV3VehicleFactory.SpawnParams({
            asset: $USDC,
            poolAddressesProvider: $AAVE_POOL_ADDRESSES_PROVIDER,
            accessControl: $sharedAC,
            feeManager: $aaveFeeManager,
            modulesManager: ModulesManager(address(0)),
            querySalt: keccak256("multilevel-aave-query"),
            deploymentSalt: keccak256("multilevel-aave-deployment"),
            forbiddenAddresses: new address[](0),
            initialExpectedSupply: _aaveExpectedSupply
        }),
        _aaveInfra
    );

    // Morpho Blue vehicle (USDC market).
    MorphoBlueVehicleInfra memory _morphoInfra = _deployMorphoBlueVehicleInfra(
        $deployer, MorphoBlueVehicleStructs.ImmutableParams({morpho: IMorpho($MORPHO)}),
        ↪  $coreFactoryRef
    );
    uint256 _morphoExpectedSupply = _previewMorphoInitialSupply($MORPHO_USDC_MARKET);
    $morphoVehicle = _deployMorphoBlueVehicleInstance(
        $deployer,
        MorphoBlueVehicleFactory.SpawnParams({
            morpho: $MORPHO,
            marketId: $MORPHO_USDC_MARKET,
            accessControl: $sharedAC,
            feeManager: $morphoFeeManager,
            modulesManager: ModulesManager(address(0)),
            querySalt: keccak256("multilevel-morpho-query"),
            deploymentSalt: keccak256("multilevel-morpho-deployment"),
            forbiddenAddresses: new address[](0),
            initialExpectedSupply: _morphoExpectedSupply
        }),
        _morphoInfra
    );
}
```

```
/* ---------------------------------------------------------------------------------------------
↪  --------------------------------- */
/*                                                    LEVEL 1
↪  */
/* ---------------------------------------------------------------------------------------------
↪  --------------------------------- */

function _deployLevel1() internal {
    $multiFeeManager = _spawnFeeManager(keccak256("multilevel-fee-manager-multi"));

    MultiVehicleInfra memory _multiInfra = _deployMultiVehicleInfra($deployer,
    ↪  $coreFactoryRef);

    MultiVehicleFactory.SpawnParams memory _params = MultiVehicleFactory.SpawnParams({
        asset: $usdc,
        name: "Multilevel",
        symbol: "ML",
        initialInterceptions: new Interceptor.Interception[](0),
        accessControl: $sharedAC,
        feeManager: $multiFeeManager,
        modulesManager: ModulesManager(address(0)),
        salts: MultiVehicleFactory.Salts({
            multiVehicle: keccak256("multilevel-multi-multiVehicle"),
            queryRedeemQueue: keccak256("multilevel-multi-queryRedeemQueue"),
            queueStrategyEngine: keccak256("multilevel-multi-queueStrategyEngine"),
            sectorAccountingEngine: keccak256("multilevel-multi-sectorAccountingEngine"),
            subQueryEngine: keccak256("multilevel-multi-subQueryEngine"),
            vehicleRegistry: keccak256("multilevel-multi-vehicleRegistry"),
            initialDepositQuery: keccak256("multilevel-multi-initialDepositQuery")
        }),
        forbiddenAddresses: new address[](0)
    });

    MultiVehicleFactory.Contracts memory _contracts =
    ↪  _deployMultiVehicleInstance($deployer, _params, _multiInfra);
    $multiVehicle = MultiVehicle(_contracts.multiVehicle);
    $strategyEngine = QueueStrategyEngine(_contracts.queueStrategyEngine);
    $accountingEngine = SectorAccountingEngine(_contracts.sectorAccountingEngine);
    $redeemQueue = QueryRedeemQueue(_contracts.queryRedeemQueue);
    $subQueryEngine = SubQueryEngine(_contracts.subQueryEngine);
    $vehicleRegistry = VehicleRegistry(_contracts.vehicleRegistry);

    // Authorize both sub-vehicles and configure a 50/50 deposit queue.
    // The MultiVehicle's own access control is `$sharedAC`; we already hold
    ↪  DEFAULT_ADMIN_ROLE there.
    $sharedAC.grantRole(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION, address(this));
    $sharedAC.grantRole(Roles.MULTI_VEHICLE_SET_QUEUES, address(this));
    $sharedAC.grantRole(Roles.MULTI_VEHICLE_DISPATCH, address(this));
    $sharedAC.grantRole(Roles.MULTI_VEHICLE_REBALANCE, address(this));
    $sharedAC.grantRole(Roles.MULTI_VEHICLE_FEED_QUERY_REDEEM_QUEUE, address(this));
    $sharedAC.grantRole(Roles.MULTI_VEHICLE_RETRIEVE_QUERY_REDEEM_QUEUE_ASSETS,
    ↪  address(this));
    $sharedAC.grantRole(Roles.MULTI_VEHICLE_PROGRESS_QUERY, address(this));
    $sharedAC.grantRole(Roles.FEE_MANAGER_SET_FEES, address(this));

    $vehicleRegistry.authorize(IVehicle(address($aaveVehicle)));
    $vehicleRegistry.authorize(IVehicle(address($morphoVehicle)));

    IQueueStrategyEngine.QueueEntry[] memory _depositQueue = new
    ↪  IQueueStrategyEngine.QueueEntry[](2);
    _depositQueue[0] = IQueueStrategyEngine.QueueEntry({
```

```
        vehicle: IBaseVehicle(address($aaveVehicle)), target: Target({value:
        ↪  type(uint256).max, threshold: 0})
    });
    _depositQueue[1] = IQueueStrategyEngine.QueueEntry({
        vehicle: IBaseVehicle(address($morphoVehicle)), target: Target({value:
        ↪  type(uint256).max, threshold: 0})
    });

    IQueueStrategyEngine.QueueEntry[] memory _redeemQueue = new
    ↪  IQueueStrategyEngine.QueueEntry[](2);
    _redeemQueue[1] = IQueueStrategyEngine.QueueEntry({
        vehicle: IBaseVehicle(address($aaveVehicle)), target: Target({value: 0, threshold:
        ↪  0})
    });
    _redeemQueue[0] = IQueueStrategyEngine.QueueEntry({
        vehicle: IBaseVehicle(address($morphoVehicle)), target: Target({value: 0,
        ↪  threshold: 0})
    });
    $strategyEngine.setQueues(_depositQueue, _redeemQueue);
}

/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */
/*                                                          LEVEL 2
↪  */
/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */

function _deployLevel2() internal {
    Conduit _impl = new Conduit();
    $conduitBeacon = new FreezablePausableBeacon(address(_impl), $sharedAC);
    $conduitFactory = new ConduitFactory($coreFactoryRef, $conduitBeacon, $sharedAC,
    ↪  _getOrDeployAssetRegistry());

    $sharedAC.grantRole(Roles.CONDUIT_SPAWN, address(this));

    // The conduit has no fee manager nor account list - the prompt only requested fees on
    ↪  the lower
    // levels. Transfer mode is permissionless so users can move conduit shares freely.
    ConduitFactory.SpawnParams memory _params = ConduitFactory.SpawnParams({
        name: "Multilevel Conduit",
        symbol: "ML-CDT",
        vehicle: IVehicle(address($multiVehicle)),
        feeManager: IFeeManager(address(0)),
        accountList: IAccountList(address(0)),
        ownerRegistry: IOwnerRegistry(address(0)),
        accessControl: $sharedAC,
        transferMode: ConduitStructs.TransferMode.ALLOW_TRANSFER,
        initialExpectedSupply: 1,
        querySalt: keccak256("multilevel-conduit-query"),
        deploymentSalt: keccak256("multilevel-conduit-deployment")
    });

    // Pre-fund the initial deposit and approve the factory.
    uint256 _initialDeposit =
    ↪  $conduitFactory.ASSET_REGISTRY().getInitialDepositAmount($USDC);
    console.log("Initial deposit amount: ", _initialDeposit);
    deal($USDC, address(this), _initialDeposit);
    IERC20($USDC).forceApprove(address($conduitFactory), _initialDeposit);

    $conduit = Conduit(address($conduitFactory.spawn(_params, _params.deploymentSalt)));
}
```

```
/* ---------------------------------------------------------------------------------------------
↪  --------------------------------- */
/*                                                 HELPERS
↪  */
/* ---------------------------------------------------------------------------------------------
↪  --------------------------------- */

/// @dev Spawns a FeeManager with the multilevel fee profile (10% perf, 1% mgmt, 0/0
↪  transactional).
function _spawnFeeManager(bytes32 salt) internal returns (FeeManager feeManager) {
    FeeManagerInfra memory _infra = _deployFeeManagerInfra($deployer, $coreFactoryRef,
    ↪  false);

    IFeeManager.FeeRecipient[] memory _recipients = new IFeeManager.FeeRecipient[](1);
    _recipients[0] = IFeeManager.FeeRecipient({target: $feeRecipient, shareBps: 10_000});

    FeeManagerFactory.SpawnParams memory _params = FeeManagerFactory.SpawnParams({
        accessControl: $sharedAC,
        initialFees: IFeeManager.Fees({
            performanceFeeBps: $PERFORMANCE_FEE_BPS,
            managementFeeBps: $MANAGEMENT_FEE_BPS,
            depositFeeBps: $DEPOSIT_FEE_BPS,
            redeemFeeBps: $REDEEM_FEE_BPS
        }),
        initialMaxFees: IFeeManager.Fees({
            performanceFeeBps: $PERFORMANCE_FEE_BPS,
            managementFeeBps: $MANAGEMENT_FEE_BPS,
            depositFeeBps: $MAX_DEPOSIT_FEE_BPS,
            redeemFeeBps: $MAX_REDEEM_FEE_BPS
        }),
        initialRecipients: _recipients,
        deploymentSalt: salt
    });

    feeManager = _deployFeeManagerInstance($deployer, _params, _infra);
}

/// @dev Replicates `AaveV3VehicleForkConfig._deployVehicle` initial-supply math so the
↪  factory's
///      slippage check `initialExpectedSupply` succeeds.
function _previewAaveInitialSupply(address asset) internal view returns (uint256) {
    uint8 _assetDecimals = IERC20Metadata(asset).decimals();
    address _pool = IPoolAddressesProvider($AAVE_POOL_ADDRESSES_PROVIDER).getPool();
    uint256 _liquidityIndex = IPool(_pool).getReserveNormalizedIncome(asset);
    uint256 _initialDepositAmount = 10 ** uint256(_assetDecimals);
    uint256 _expectedScaledShares =
    ↪  TokenMath.getATokenMintScaledAmount(_initialDepositAmount, _liquidityIndex);
    return SharesLib.scale(_expectedScaledShares, _assetDecimals, 18, Math.Rounding.Floor);
}

/// @dev Replicates `MorphoBlueVehicleForkConfig._deployVehicle` initial-supply math:
↪  simulate a 1-token
///      supply against the live market, take the resulting supplyShares delta, and scale
↪  to 18 decimals.
function _previewMorphoInitialSupply(MarketId marketId) internal returns (uint256) {
    MarketParams memory _params = IMorpho($MORPHO).idToMarketParams(marketId);
    uint8 _assetDecimals = IERC20Metadata(_params.loanToken).decimals();
    uint256 _initialDepositAmount = 10 ** uint256(_assetDecimals);

    uint256 _snap = vm.snapshotState();
    deal(_params.loanToken, address(this), _initialDepositAmount);
```

```
        IERC20(_params.loanToken).forceApprove($MORPHO, _initialDepositAmount);
        uint256 _supplySharesBefore = IMorpho($MORPHO).position(marketId,
        ↪  address(this)).supplyShares;
        IMorpho($MORPHO).supply(_params, _initialDepositAmount, 0, address(this), hex"");
        uint256 _morphoSharesReceived =
            IMorpho($MORPHO).position(marketId, address(this)).supplyShares -
            ↪  _supplySharesBefore;
        vm.revertToState(_snap);

        return SharesLib.scale(_morphoSharesReceived, _assetDecimals + 6, 18,
        ↪  Math.Rounding.Floor);
    }
}

/// @title MultilevelRailnet_MainnetFork
/// @notice Smoke test that verifies the multilevel Railnet stack (Aave + Morpho vehicles ⊂
↪  MultiVehicle ⊂ Conduit)
///         can be deployed against a real mainnet fork and that a user can:
///         1. Deposit USDC into the Conduit and receive conduit shares.
///         2. See the funds dispatched into both Aave and Morpho through the MultiVehicle's
↪  strategy queue.
///         3. Observe fee accounting (10% perf / 1% mgmt) reflected on every layer that owns a
↪  FeeManager.
contract MultilevelRailnet_MainnetFork is MultilevelRailnetForkConfig, ForkUtils {
    function setUp() public {
        _startFork("MAINNET");
        _deployAll();
    }

    function test_PoC_requestWithdrawableUsesStaleTotalAssetsForQueuedRedeems() public {
        $multiFeeManager.setFees(
            IFeeManager.Fees({performanceFeeBps: 0, managementFeeBps: 0, depositFeeBps: 0,
            ↪  redeemFeeBps: 0})
        );
        $aaveFeeManager.setFees(
            IFeeManager.Fees({
                performanceFeeBps: 0, managementFeeBps: 0, depositFeeBps: 0, redeemFeeBps:
                ↪  $MAX_REDEEM_FEE_BPS
            })
        );
        _setAaveOnlyQueues(false);

        address _queuedRedeemer = makeAddr("ml-queued-redeemer");
        address _triggerRedeemer = makeAddr("ml-trigger-redeemer");
        uint256 _amount = 10_000 * 10 ** $usdc.decimals();

        _depositIntoMultiVehicle(_queuedRedeemer, _amount,
        ↪  keccak256("stale-ta-deposit-queued"));
        _depositIntoMultiVehicle(_triggerRedeemer, _amount,
        ↪  keccak256("stale-ta-deposit-trigger"));

        uint256 _queuedShares = IERC20(address($multiVehicle)).balanceOf(_queuedRedeemer);
        uint256 _triggerShares = IERC20(address($multiVehicle)).balanceOf(_triggerRedeemer);
        // approximately equal depending on the underlying asset precision (18 - 6 = 12)
        assertApproxEqAbs(
            _queuedShares / (10 ** 12), _triggerShares / (10 ** 12), 1, "equal deposits should
            ↪  mint equal shares"
        );
        assertEq($accountingEngine.withdrawable(), 0, "all deposits should be allocated to
        ↪  Aave");

        Query memory _queuedRedeem =
```

```
        _buildMultiVehicleRedeemQuery(_queuedRedeemer, _queuedShares,
        ↪  keccak256("stale-ta-redeem-queued"));
    vm.prank(_queuedRedeemer);
    State _queuedState = $multiVehicle.create(_queuedRedeem);
    assertEq(uint8(_queuedState), uint8(State.PROCESSING), "first redeem should be
    ↪  queued");
    assertEq($redeemQueue.demandsCount(), 1, "queued redeem demand not created");
    assertEq($redeemQueue.unredeemable(), _queuedShares, "queued shares mismatch");

    _setAaveOnlyQueues(true);

    uint256 _totalAssetsBefore = $multiVehicle.totalAssets();
    uint256 _aaveRedeemFeesBefore = $usdc.balanceOf(address($aaveFeeManager));

    Query memory _triggerRedeem =
        _buildMultiVehicleRedeemQuery(_triggerRedeemer, _triggerShares,
        ↪  keccak256("stale-ta-redeem-trigger"));
    vm.startPrank(_triggerRedeemer);
    State _triggerState = $multiVehicle.create(_triggerRedeem);
    assertEq(uint8(_triggerState), uint8(State.UNLOCKING), "second redeem should unlock
    ↪  immediate liquidity");
    $multiVehicle.unlock(_triggerRedeem);
    vm.stopPrank();

    vm.prank(_queuedRedeemer);
    $multiVehicle.unlock(_queuedRedeem);

    uint256 _aaveRedeemFees = $usdc.balanceOf(address($aaveFeeManager)) -
    ↪  _aaveRedeemFeesBefore;
    assertGt(_aaveRedeemFees, 0, "Aave redeem fee must be charged during
    ↪  requestWithdrawable");
    assertLt(
        $multiVehicle.totalAssets(),
        _totalAssetsBefore - _aaveRedeemFees / 2,
        "requestWithdrawable must reduce totalAssets when redeem fees are paid"
    );

    uint256 _queuedRedeemed = $usdc.balanceOf(_queuedRedeemer);
    uint256 _triggerRedeemed = $usdc.balanceOf(_triggerRedeemer);
    (uint256 _multiAaveShares,,,,) =
    ↪  $accountingEngine.vehicleHoldings(IBaseVehicle(address($aaveVehicle)));

    console.log("Queued redeemer USDC:  ", _queuedRedeemed);
    console.log("Trigger redeemer USDC: ", _triggerRedeemed);
    console.log("Aave redeem fee USDC:  ", _aaveRedeemFees);
    console.log("MV Aave shares:        ", _multiAaveShares);
    assertEq(_multiAaveShares, 0, "all shares should have been redeemed from Aave to
    ↪  fulfill the redeems");

    // @audit these two should be approximately equal! Uncommenting the following assertion
    ↪  reverts
    /* assertApproxEqAbs(
        _queuedRedeemed,
        _triggerRedeemed,
        10,
        "equal redeemers should receive the same amount after the shared redeem fee"
    ); */
}

/* ----------------------------------------------------------------------------------------
↪  ------------------------------- */
```

```
/*                                                                HELPERS
↪  */
/* ---------------------------------------------------------------------------------------
↪  ---------------------------------- */

function _assertFees(IFeeManager feeManager, string memory label) internal view {
    (, IFeeManager.Fees memory _fees) = feeManager.fees();
    assertEq(_fees.performanceFeeBps, $PERFORMANCE_FEE_BPS, string.concat(label, ":
    ↪  performanceFeeBps"));
    assertEq(_fees.managementFeeBps, $MANAGEMENT_FEE_BPS, string.concat(label, ":
    ↪  managementFeeBps"));
    assertEq(_fees.depositFeeBps, $DEPOSIT_FEE_BPS, string.concat(label, ":
    ↪  depositFeeBps"));
    assertEq(_fees.redeemFeeBps, $REDEEM_FEE_BPS, string.concat(label, ": redeemFeeBps"));
}

/// @dev Initial deposit performed by `MultiVehicleFactory.spawn` (1 USDC). Captured at
↪  first call to keep
///      reuse cheap across multiple assertions in the same test.
function _initialMultiVehicleAssets() internal view returns (uint256) {
    return 10 ** $usdc.decimals();
}

function _setPerformanceOnlyFees(IFeeManager feeManager) internal {
    feeManager.setFees(
        IFeeManager.Fees({
            performanceFeeBps: $PERFORMANCE_FEE_BPS,
            managementFeeBps: 0,
            depositFeeBps: $DEPOSIT_FEE_BPS,
            redeemFeeBps: $REDEEM_FEE_BPS
        })
    );
}

function _spawnFeeEnabledConduit(IFeeManager feeManager) internal returns (Conduit
↪  conduit) {
    ConduitFactory.SpawnParams memory _params = ConduitFactory.SpawnParams({
        name: "Performance Fee Conduit",
        symbol: "PFC",
        vehicle: IVehicle(address($multiVehicle)),
        feeManager: feeManager,
        accountList: IAccountList(address(0)),
        ownerRegistry: IOwnerRegistry(address(0)),
        accessControl: $sharedAC,
        transferMode: ConduitStructs.TransferMode.ALLOW_TRANSFER,
        initialExpectedSupply: 1,
        querySalt: keccak256("multilevel-conduit-fee-query"),
        deploymentSalt: keccak256("multilevel-conduit-fee-deployment")
    });

    uint256 _initialDeposit =
    ↪  $conduitFactory.ASSET_REGISTRY().getInitialDepositAmount(address($usdc));
    deal(address($usdc), address(this), _initialDeposit);
    IERC20(address($usdc)).approve(address($conduitFactory), 0);
    IERC20(address($usdc)).approve(address($conduitFactory), _initialDeposit);

    conduit = Conduit(address($conduitFactory.spawn(_params, _params.deploymentSalt)));
}

function _estimateConduitRedeem(Conduit conduit, uint256 conduitShares) internal view
↪  returns (uint256) {
    Asset[] memory _input = new Asset[](1);
```

```
    _input[0] = Asset({asset: address(conduit), value: conduitShares});
    Asset[] memory _output = conduit.estimate(_input, Mode.REDEEM, EstimationType.OUTPUT);
    return _output[0].value;
}

function _setMorphoOnlyQueues() internal {
    IQueueStrategyEngine.QueueEntry[] memory _depositQueue = new
    ↪  IQueueStrategyEngine.QueueEntry[](1);
    _depositQueue[0] = IQueueStrategyEngine.QueueEntry({
        vehicle: IBaseVehicle(address($morphoVehicle)), target: Target({value:
        ↪  type(uint256).max, threshold: 0})
    });

    IQueueStrategyEngine.QueueEntry[] memory _redeemQueue = new
    ↪  IQueueStrategyEngine.QueueEntry[](1);
    _redeemQueue[0] = IQueueStrategyEngine.QueueEntry({
        vehicle: IBaseVehicle(address($morphoVehicle)), target: Target({value: 0,
        ↪  threshold: 0})
    });

    $strategyEngine.setQueues(_depositQueue, _redeemQueue);
}

function _setAaveOnlyQueues(bool redeemEnabled) internal {
    IQueueStrategyEngine.QueueEntry[] memory _depositQueue = new
    ↪  IQueueStrategyEngine.QueueEntry[](1);
    _depositQueue[0] = IQueueStrategyEngine.QueueEntry({
        vehicle: IBaseVehicle(address($aaveVehicle)), target: Target({value:
        ↪  type(uint256).max, threshold: 0})
    });

    IQueueStrategyEngine.QueueEntry[] memory _redeemQueue =
        new IQueueStrategyEngine.QueueEntry[](redeemEnabled ? 1 : 0);
    if (redeemEnabled) {
        _redeemQueue[0] = IQueueStrategyEngine.QueueEntry({
            vehicle: IBaseVehicle(address($aaveVehicle)), target: Target({value: 0,
            ↪  threshold: 0})
        });
    }

    $strategyEngine.setQueues(_depositQueue, _redeemQueue);
}

function _buildMultiVehicleRedeemQuery(address user, uint256 shares, bytes32 salt)
    internal
    view
    returns (Query memory redeemQuery)
{
    Asset[] memory _input = new Asset[](1);
    _input[0] = Asset({asset: address($multiVehicle), value: shares});
    Asset[] memory _output = $multiVehicle.estimate(_input, Mode.REDEEM,
    ↪  EstimationType.OUTPUT);

    redeemQuery = Query({
        owner: user, receiver: user, input: _input, output: _output, mode: Mode.REDEEM,
        ↪  salt: salt, data: ""
    });
}

function _depositIntoMultiVehicle(address user, uint256 amount, bytes32 salt) internal {
    deal(address($usdc), user, amount);
```

```
        Asset[] memory _input = new Asset[](1);
        _input[0] = Asset({asset: address($usdc), value: amount});
        Asset[] memory _output = $multiVehicle.estimate(_input, Mode.DEPOSIT,
        ↪  EstimationType.OUTPUT);

        Query memory _depositQuery = Query({
            owner: user, receiver: user, input: _input, output: _output, mode: Mode.DEPOSIT,
            ↪  salt: salt, data: ""
        });

        vm.startPrank(user);
        IERC20(address($usdc)).approve(address($multiVehicle), amount);
        $multiVehicle.create(_depositQuery);
        $multiVehicle.unlock(_depositQuery);
        vm.stopPrank();
    }

    function _redeemFromMultiVehicle(address user, uint256 shares, bytes32 salt) internal {
        Asset[] memory _input = new Asset[](1);
        _input[0] = Asset({asset: address($multiVehicle), value: shares});
        Asset[] memory _output = $multiVehicle.estimate(_input, Mode.REDEEM,
        ↪  EstimationType.OUTPUT);

        Query memory _redeemQuery = Query({
            owner: user, receiver: user, input: _input, output: _output, mode: Mode.REDEEM,
            ↪  salt: salt, data: ""
        });

        vm.startPrank(user);
        IERC20(address($multiVehicle)).approve(address($multiVehicle), shares);
        $multiVehicle.create(_redeemQuery);
        $multiVehicle.unlock(_redeemQuery);
        vm.stopPrank();
    }

    function _estimateMultiVehicleRedeem(uint256 shares) internal view returns (uint256) {
        Asset[] memory _input = new Asset[](1);
        _input[0] = Asset({asset: address($multiVehicle), value: shares});
        Asset[] memory _output = $multiVehicle.estimate(_input, Mode.REDEEM,
        ↪  EstimationType.OUTPUT);
        return _output[0].value;
    }
}
```

Recommendation: The total assets value must be updated after every operation that will modify its value. `totalAssets()` should be called before `__tryAutoRedemption` to ensure that the value is up-to-date.

Kiln: Fixed in commit b5604ad6.

Fixed by applying the recommendation. `_createRedeem` refreshes `totalAssets_` after `requestWithdrawable()` (and recomputes queue demands via `exitSupplies`) before `__tryAutoRedemption` consumes it, so queue-fulfillment rate reflects any sub-vehicle redeem fees crystallized during the dispatch path.

Spearbit: Fixed. The `totalAssets_` value is correctly updated. Tests show that the issue is fixed.

6.1.6 Automatic allocations to vehicles with deposit fees break share fairness

Severity: Medium Risk.

Context: SectorAccountingEngine.sol#L782-L789.

Description: The MultiVehicle mints shares based on the gross amount of assets supplied by the current deposit query. The assets are later allocated into child vehicles where deposit fees can reduce the amount of child-vehicle shares actually received by the MultiVehicle.

The MV share rate is effectively based on `totalAssets / totalSupply`. When a user deposits, MultiVehicle calculates the shares to mint `sharesToMint = assetsToDeposit * totalSupply / totalAssets`. This calculation uses the full `assetsToDeposit` value from the current query. After that, the funds are sent to the `SectorAccountingEngine`, which may allocate them into an underlying vehicle. If that child vehicle has a deposit fee, the fee is applied to the child vehicle output shares during unlock, so the MultiVehicle receives only the net post-fee child shares which represent less assets than what is deposited.

As a result, the MV supply increases as if the full deposit amount entered the portfolio, while MV assets increase only by the net amount after the child vehicle deposit fee. The missing value is reflected as a lower MV share rate. Because the share rate is global, the loss is spread across **all MV shareholders**, including holders who did not initiate the deposit. The deposit fee is therefore socialized between existing holders and the new depositor, even though the funds being deposited correspond only to the current query owner's funds.

This is unfair to share holders. A new depositor can repeatedly deposit and redeem through the MultiVehicle path, causing each child-vehicle deposit fee to reduce the global MV share price. Existing holders absorb part of each fee through dilution, even though they did not perform the fee-incurring action.

Proof of Concept: The proof of concept example is that:

- At block N: user 1 deposits 10k USDC and receive X shares.
  - Automatic allocation to underlying vehicle ⇒ fee is taken.
- At block N: user 2 deposits 10k USDC. Share calculations result in Y shares (greater than X).
  - Automatic allocation to underlying vehicle ⇒ fee is taken (both user 1 and user 2 are affected by this fee).

See the high difference in "Estimated redeem output".

```
[PASS] test_DepositFeesBreakShareFairnessMVOnly() (gas: 5507840)
Logs:
  Forking at block number from env: 25071042
  AAVE initial deposit amount:  1000000
  Morpho initial deposit amount:  1000000
  Initial deposit amount:  1000000
  --------------------    Initial state     --------------------
  MV      total assets:   1999999
  Aave    total assets:   2999998
  Morpho  total assets:   999999

  MV fee manager address:   0x6c65a811BD0eC01799f6a6A8BCdDe17A0e2073C2
  Fee Mgr balance in MV :   0

  --------------------    User1: Deposit 10k USDC     --------------------
  MV      total assets:   9001999998
  Aave    total assets:   10002999998
  Morpho  total assets:   999999

  User shares in MV:   10000005000002500001250
  Estimated redeem output:  9000199958

  --------------------    User2: Deposit and withdraw 10k USDC     --------------------
  User1 shares in Conduit:   10000005000002500001250
  User2 shares in Conduit:   11110869809180930863237
  User 1 - Estimated redeem output:  8526555081
  User 2 - Estimated redeem output:  9473739606
```

```
MV      total assets:   8528260392
Aave    total assets:   10529260391
Morpho  total assets:   999999

User2 shares in Conduit:   0

--------------------    Final situation     --------------------
User1 shares in Conduit:   10000005000002500001250
User2 shares in Conduit:   0
User 1 - Estimated redeem output:  8526555081
MV      total assets:   1705310
Aave    total assets:   2002705309
Morpho  total assets:   999999

User1 USDC balance:   8526555081  // @audit user1 has way less assets than user2
User2 USDC balance:   9473739606
```

User 1 will receive 8,526 USDC after depositing 10k USDC, while user 2 will receive 9,473 USDC. **This is highly unfair**.

Recommendation: One approach to fix this issue is to never use automatic allocations with vehicles that have a deposit fee set.

Alternatively, another approach would be to account for the deposited amount after allocation is executed. However, fairness could be broken depending on which type of vehicles funds are currently deposited to. For example when having two automatic allocations (a vehicle with no deposit fee, and a second vehicle with a deposit fee), users will prefer the one without fees.

(Find the Proof of Concept code in the Appendix).

Kiln: Fixed in commit 77d03ca7 and commit 86337139. Additional fix in commit a19e548.

Fixed by applying the second alternative. `_createDeposit` has been rewritten with a delta-based mint: it snapshots `totalAssets()` before and after `accountingEngine.deposit(...)`, takes `_delta = min(after - before, _assetsLeft)`, and mints `previewDeposit(_delta, oldSupply, oldAssets)` at the pre-deposit rate. Existing holders are not diluted by a lossy sub-vehicle. The loss is borne entirely by the incoming depositor. Commit 86337139 adds a post-mint slippage re-check that reverts `InvalidEstimation` when the minted share count falls below the user's `query.output[0].value` floor, so the depositor's slippage protection survives the lossy path.

Spearbit: Verified the fixes for the original issue: tests show that share fairness is kept and the difference in total assets after the deposit into the underlying vehicles is now accounted to the depositor. Kiln also submitted an additional fix (commit a19e548d) for a follow-up item listed in the appendix (see the "Focus point document"); its review was not completed during the engagement and it requires further verification.

6.2 Low Risk

6.2.1 Improve the `VehicleLib.isCategory` validation for the `SingleAsset` category

Severity: Low Risk.

Description: The `VehicleLib.isCategory` function is not validating the deposit and redeem routes of the vehicle in case the tested category is `SingleAsset`.

If the vehicle must be a single-asset vehicle, it **must** have only exactly one deposit and one redeem route.

Recommendation: Kiln should add the following additional validation to the function.

```
if( testedCategory == VehicleCategory.SingleAsset && (_depositRoutes.length != 1 ||
↪  _redeemRoutes.length != 1) ) {
    return false;
}
```

Kiln: Fixed in commit 6525639c (`isCategory` SingleAsset route check).

Fixed by applying the recommendation — `VehicleLib.isCategory` rejects SingleAsset when deposit or redeem routes ≠ exactly 1.

Spearbit: Verified the fixes.

6.2.2 `BaseVehicle` functions `maxDeposit` and `maxRedeem` should return zero when the `create` function cannot be called

Severity: Low Risk.

Description: There are multiple cases where the `BaseVehicle.create` function might revert. Some of them can be extrapolated to be used in the `maxDeposit` and `maxRedeem` functions to return 0.

- `$.forbiddenAddresses[msg.sender] == true` (note that `query.owner` must be `msg.sender` because of the `_onlyQueryOwner` validation).
- `!_ready() && msg.sender != $.deployer`
- The access manager is configured but the `msg.sender` does not have the expected role given the `query.mode` (`Roles.VEHICLE_STEAM_DEPOSIT` for deposits, `Roles.VEHICLE_STEAM_REDEEM` for redeems).

Recommendation: Kiln should return `0` when `maxDeposit` is executed and `account` violates one of the following checks.

- `$.forbiddenAddresses[account] == true`
- `!_ready() && account != BaseVehicleStore.getStorage().deployer`
- `BaseVehicleStore.getStorage().accessControl != address(0) && BaseVehicleStore.getStorage().enabled && BaseVehicleStore.getStorage().accessControl.hasRoleOrScopedRole(Roles.VEHICLE_STEAM_DEPOSIT, address(this), account) == false`

Kiln should return `0` when `maxRedeem` is executed and `account` violates one of the following checks.

- `$.forbiddenAddresses[account] == true`
- `!_ready() && account != BaseVehicleStore.getStorage().deployer`
- `BaseVehicleStore.getStorage().accessControl != address(0) && BaseVehicleStore.getStorage().enabled && BaseVehicleStore.getStorage().accessControl.hasRoleOrScopedRole(Roles.VEHICLE_STEAM_REDEEM, address(this), account) == false`

Kiln: Acknowledged.

`maxDeposit` / `maxRedeem` are advisory capacity views; the authoritative authorization layer is the `create()`-time gates. Integrators must handle the `create()` revert paths.

6.2.3 Avoid deleting the subquery data once it reaches the final state

Severity: Low Risk.

Description: The current logic of the `SubQueryEngine` resets the subquery data (`$.subQueries[_subQueryId] = 0;`) once it has reached the final `SETTLED` or `REJECTED` state.

Subqueries are bound to a Query which makes them unique by default because.

1) The `createSubQuery` function reverts if a Query has already been "registered".
2) The `SubQuery` struct includes the `queryId` in its data structure, so the `subQueryId` is also unique (because of Point 1, you can't register multiple queries).

This means that both Query and Subquery are unique and once registered they cannot be reused. By "resetting" the `subQueryId` you lose important information like the "end state" of the query.

Instead of resetting them, we could for example track the final state to be later on used by the `subqueryStatus` function.

```
- mapping(Id => uint248) subQueries;
+ mapping(Id => SubQueryState) subQueries;

+ struct SubQueryState {
+     uint248 index;
+     State finalState; // EMPTY -> `index == 0` -> never created, `index > 0` -> still
↪  ongoing, not ended yet
+ }

  function progressQuery(SubQuery calldata subQuery, Query calldata query)
      external
      nonReentrant
      onlyAccountingEngineOrRole(Roles.MULTI_VEHICLE_PROGRESS_QUERY)
      returns (State queryState)
  {
      // other code

      if (queryState == State.SETTLED || queryState == State.REJECTED) {
          // other code

-          $.subQueries[_subQueryId] = 0;
+          $.subQueries[_subQueryId].finalState = queryState;
      }

      // other code
  }
```

By avoiding the reset behavior and setting the final state once we can both improve the DX and fix unexpected behaviors.

1) The next time we enter `progressQuery` instead of reverting with `UnknownSubQuery` we can provide a better custom error like `SubQueryAlreadyEnded($.subQueries[_subQueryId].finalState)` .
2) We can return the **correct** subquery state when `subqueryStatus` is executed.

```
  function subqueryStatus(SubQuery calldata subQuery, Query calldata query)
      external
      view
      returns (KeeperLib.JobStatus)
  {
      Id _subQueryId = subQuery.toId(address(this));
      SubQueryEngineStorage storage $ = _getStorage();
```

```
      if ($.subQueries[_subQueryId] == 0) {
          revert UnknownSubQuery(_subQueryId);
      }
+     if ($.subQueries[_subQueryId].finalState != State.EMPTY) {
+         return KeeperLib.JobStatus.STOP;
+     }
      State _queryState = subQuery.vehicle.state(query);
      if (_queryState == State.PAUSED || _queryState == State.PROCESSING) {
          return KeeperLib.JobStatus.POLL;
      }
      return KeeperLib.JobStatus.EXEC;
  }
```

Recommendation: Kiln should consider implementing the suggestions made above.

Kiln: Fixed in commit a2ff4efd.

Fixed by applying the recommendation — `SubQueryState` struct preserves `finalState`; `progressQuery` reverts `SubQueryAlreadyFinalized(id, state)`; `subQueryStatus` returns `STOP` for terminal queries.

Spearbit: Verified the fixes. The final state of a sub-query is preserved and `progressQuery` reverts on already finalized sub-queries.

6.2.4 `SubQueryEngine.subqueryStatus` does not validate the query-subquery relationship

Severity: Low Risk.

Description: `SubQueryEngine.subqueryStatus` function **does not** validate that the subquery is indeed bound to the query (there's a 1:1 relationship between query and subquery).

The function should execute the following validation.

```
Id _queryId = query.toId(address(subQuery.vehicle));
if (!subQuery.queryId.eq(_queryId)) {
    revert QueryMismatch(_queryId, subQuery.queryId);
}
```

Recommendation: Kiln should implement the suggestion reported above.

Kiln: Fixed in commit 750f9277.

Fixed by applying the recommendation — `subQueryStatus` validates the query/sub-query binding and reverts `QueryMismatch(expected, actual)` on mismatch.

Spearbit: Verified the fixes. `subQueryStatus` now validates the query and sub-query binding and reverts with `QueryMismatch`.

6.2.5 Improve the sanity checks of the `QueueStrategyEngine.setQueues`

Severity: Low Risk.

Description: The current implementation of the `QueueStrategyEngine.setQueues` does not perform any sanity checks on the vehicles added as deposit or redeem items in the `QueueStrategyEngine` contract.

After reviewing the requirements we are suggesting the following validations:

Common, to be applied to both deposit/redeem queue items:

- Revert if `queue[_idx].vehicle.asset() != $.accountingEngine.asset()`: the vehicle must be compatible with the `MultiVehicle` and `SectorAccountingEngine` contracts.
- Revert if `queue[_idx].vehicle.ready() == FALSE`: revert if the vehicle is not ready. Deposits/redeems operations on non-ready vehicles would be skipped by the `allocate` operation or directly revert when created on the underlying vehicle.

Deposit specific:

- Revert if `$.accountingEngine.isAuthorizedVehicle(queue[_idx].vehicle) == FALSE`: an unauthorized vehicle would be skipped by the `allocate` function.
- Revert if `$.accountingEngine.asset()` is **incompatible** with one of the deposit routes of `queue[_idx].vehicle`.
- Revert if `$.accountingEngine.getVehicleConfig(queue[_idx].vehicle).mode == VehicleRegistry.VehicleMode.Manual`: the allocate function **always** skips the vehicle in that case.
- Revert if `queue[_idx].target.value == 0`: the allocate function will always skip the vehicle. `_currentQueueEntry.target.constrainBy` will return the queue's target and `_effectiveTarget.isAtTarget` will always return `true`.

Redeem specific:

- Revert if `queue[_idx].target.upperBound() == Target.UNLIMITED()`: the unallocate function **always** skips the vehicle in that case.
- Revert if `VehicleLib.isCategory(queue[_idx].vehicle, VehicleLib.VehicleCategory.SingleAsset)`.
- Revert if `$.accountingEngine.asset()` is **incompatible** with one of the redeem routes of `queue[_idx].vehicle`.

Some of the above checks are always valid (have no external state dependencies), and some of them are ”pre-checks” that could change depending on the vehicle or the `VehicleRegistry` state but they still make sense at the time of the `setQueues` execution.

Recommendation: Kiln should implement all the above sanity checks depending on the queue type (deposit/redeem). Natspec comments may require updates.

Kiln: Fixed in commit 6fea57ab.

Fixed by applying the recommendation — new `_checkQueue` enforces every prescribed condition (asset mismatch, not ready, unauthorized, Manual mode, zero target, non-SingleAsset, unlimited redeem target, plus the F17 redeem-side Manual check) at `setQueues` time, reverting `InvalidQueueEntry(idx, vehicle, reason)`.

Spearbit: Verified the fixes. `setQueues` now validates every queue entry and reverts with `InvalidQueueEntry`.

6.2.6 `QueueStrategyEngine.allocate` could allocate more than the cap target

Severity: Low Risk.

Description: The current logic of the `QueueStrategyEngine.allocate` function iterates over the configured `depositQueue` items and skips the iteration when the effective target (configured for that item) is not unlimited and the existing vehicle's holding is already over such target.

```
    if (!_effectiveTarget.isUnlimited()) {
        // the account engine returns the full accounting details related to the vehicle
        (
            // shares that have been properly unlocked and retrieved by the engine
            uint256 _sharesAfterUnlocks,,
            uint256 _expectedSharesAfterUnlocks,,
            // assets that are going to be using in deposit queries (pre-create)
            uint256 _assetsBeforeCreates
        ) = _accountingEngine.vehicleHoldings(_currentQueueEntry.vehicle);

        // we start by including the actively held shares.
        _vehicleSharesHoldings = _sharesAfterUnlocks;

        // we also include and estimate assets that are about to be deposited into shares, if
        ↪  any
        if (_assetsBeforeCreates > 0) {
            _vehicleSharesHoldings += _currentQueueEntry.vehicle
            .estimateSingleAssetShares(_assetsBeforeCreates, true);
        }

        // we then include expected shares from processing deposits
        _vehicleSharesHoldings += _expectedSharesAfterUnlocks;

>>>         // 3. Skip if at target (within threshold tolerance)
>>>         if (_effectiveTarget.isAtTarget(_vehicleSharesHoldings)) {
>>>             continue;
>>>         }
    }
```

This logic does not consider three important facts:

- The `depositQueue` can contain duplicates of the same vehicle.
- The `allocate` function is executed **before** the actual deposits.
- The `allocate` function is not considering the ”hypothetical” deposit performed by previous iterations on the same (duplicate) vehicle.

Because of the above facts it's possible that a Deposit operation could end up holding more shares of a vehicle compared to the allowed upper bound target configured by the Asset Manager in the `QueueStrategyEngine`.

Recommendation: Kiln has two options:

- Avoid allowing the `depositQueue` to contain duplicates.
- Refactor the `QueueStrategyEngine.allocate` logic to include in the `_vehicleSharesHoldings` the ”hypothetical” shares obtained by a deposit allowed in the previous iterations for the same vehicle.

Kiln: Fixed in commit c8eea0e5 (`VehicleCommitment` accumulator, shared with the `unallocate` finding).

Fixed by applying recommendation #2 — `VehicleCommitment` accumulator added to `allocate` so duplicate vehicles correctly subtract prior in-call commitments from their per-vehicle caps. Symmetric to the unallocate-side accumulator that prompted the commit subject tag.

Spearbit: The implementation provided by the commit c8eea0e5 does not consider the prior share deposited when `if (_effectiveTarget.isAtTarget(_vehicleSharesHoldings)) {` is executed. Because of that the `a`

`llocate` function could allocate more shares than the one that should be allowed by the `_effectiveTarget`. For a more detailed explanation see the Cantina comment QueueStrategyEngine.sol?preview=code&comment_-id=c5d7a37e-b3f9-4e42-be6f-380b82e3368f. Partially resolved. After the engagement Kiln submitted a follow-up fix (PR 446, commit eb5a540) that makes the holdings commitment-aware before the `isAtTarget` check; it has not been reviewed by Spearbit and is listed in the appendix (see the "Focus point document") as requiring further verification.

6.2.7 `QueueStrategyEngine` functions `allocate` and `unallocate` should "simulate" the outcome to avoid reverts

Severity: Low Risk.

Description: Both the `allocate` and `unallocate` functions do not "simulate" the outcome of the final operation of the filtered `Allocation` returned.

The final deposit or redeem actions performed on the underlying vehicle could revert, making the whole root transaction revert as a consequence.

Recommendation: Kiln should consider skipping the vehicle's allocation and unallocation if the final operation performed on the vehicle would result in a revert.

It's important to note that when the actual dispatch function is executed in the `SectorAccountingEngine`, the vehicle's sector could already contain some assets/shares that would be included in the operation on top of the amount calculated in the `Allocation` returned by the `allocate` and `unallocate`.

This means that even if the "simulated" allocation/unallocation would fail when evaluated in the `QueueStrategy Engine` loop, the actual operation could instead succeed.

Kiln: Acknowledged.

A try/catch simulation per iteration would add a full deposit/redeem dry-run cost per queue entry — substantial gas. The practical revert window is meaningfully narrowed by collateral fixes: commit c8eea0e5 (`VehicleCommitment` accumulator in `allocate` + `unallocate`) prevents the duplicate-vehicle double-counting that was a primary source of estimate-vs-reality drift; commit 6fea57ab (`setQueues` sanity) catches misconfigured entries at install time; commit 9ae74f3a (clamp `maxDeposit` / `maxRedeem` to sector balance) seals dispatch against donation-inflated capacity reads. The residual case — an estimate that becomes stale between QSE planning and SAE dispatch (e.g. sub-vehicle fee spike) — is accepted as an operational concern; the asset manager removes the misbehaving vehicle from the queue.

Spearbit: Acknowledged.

6.2.8 `QueueStrategyEngine.unallocate` should not allow unallocations for vehicle's configured with "manual" operating mode

Severity: Low Risk.

Description: The current `QueueStrategyEngine.unallocate` does not perform any validation against the Operating Mode configured for the vehicle in the `VehicleRegistry`.

If the `vehicle` is configured in the `VehicleRegistry` and the `mode` is equal to `VehicleRegistry.Vehicle Mode.Manual`, the iteration should be skipped.

Note that if the vehicle is not configured in the `VehicleRegistry` it means that it has never been authorized or has been unauthorized. In that case the `_accountingEngine.getVehicleConfig(_currentQueueEntry.vehicle) .mode` would return the "default" empty value of the `enum VehicleMode` which currently is `Automatic`.

Recommendation: Kiln should skip the `vehicle` in the `unallocate` loop if `_accountingEngine.getVehicleConfig(_currentQueueEntry.vehicle).mode == VehicleRegistry.VehicleMode.Manual`.

Kiln: Fixed in commit 6fea57ab.

`_checkQueue`'s redeem branch now rejects Manual-mode vehicles at install time (`redeem.manualMode`), mirroring the existing `deposit.manualMode` guard. The runtime `unallocate` loop also skips Manual vehicles, mirroring the `allocate` runtime check — covers the lazy-revocation path where a vehicle is reconfigured to Manual after `setQueues`. Tests in `MultiVehicle.QueueStrategyEngine.t.sol` and `MultiVehicle.VehicleConfigEnforcement.t.sol`.

Spearbit: Verified the fixes. Manual-mode vehicles are rejected at `setQueues` time and skipped by `unallocate`.

6.2.9 `QueueStrategyEngine.unallocate` could unallocate more than the cap target upper bound

Severity: Low Risk.

Description: The current logic of the `QueueStrategyEngine.unallocate` function iterates over the configured `redeemQueue` items and skips the iteration when the `_vehicleSharesHoldings` are already below the `_currentQueueEntry.target.upperBound()`.

```
(uint256 _vehicleSharesHoldings,,,,) =
↪  _accountingEngine.vehicleHoldings(_currentQueueEntry.vehicle);

// 2. we check if the vehicle is not already under the target (with threshold tolerance),
↪  otherwise we skip it

uint256 _effectiveTarget = _currentQueueEntry.target.upperBound();
if (_vehicleSharesHoldings <= _effectiveTarget) {
    continue;
}
```

This logic does not consider three important facts:

- The `redeemQueue` can contain duplicates of the same vehicle.
- The `unallocate` function is executed before the actual redeem operation.
- The `unallocate` function is not considering the ”hypothetical” redeems performed by previous iterations on the same (duplicate) vehicle.

Because of the above facts it's possible that a Redeem operation could end up bringing the `_vehicleSharesHoldings` shares of a vehicle below the expected target upper bound target configured by the Asset Manager in the `QueueStrategyEngine`.

Recommendation: Kiln has two options:

- Avoid allowing the `redeemQueue` to contain duplicates.
- Refactor the `QueueStrategyEngine.unallocate` logic to remove from the `_vehicleSharesHoldings` the ”hypothetical” shares redeemed (burned) by a redeem operation allowed in the previous iterations for the same vehicle.

Kiln: Fixed in commit c8eea0e5 (track per-vehicle hypothetical redemptions in QSE.unallocate (18)).

Fixed by applying recommendation #2 — `VehicleCommitment` accumulator added to `unallocate` so duplicate vehicles correctly subtract prior in-call commitments from `_maxRedeemableShares` and `_deltaSharesToTarget`.

Spearbit: The implementation provided by commit c8eea0e5 does not consider the prior shares redeemed when the check `if (_vehicleSharesHoldings <= _effectiveTarget)` is executed, so `unallocate` could redeem more shares than the `_effectiveTarget` allows. Partially resolved: after the engagement Kiln submitted a follow-up fix (PR 446, commit 8cf4bdf) that deducts prior in-call commitments before the skip check; it has not been reviewed by Spearbit and is listed in the appendix (see the "Focus point document") as requiring further verification.

6.2.10 Additional validations across the `QueryRedeemQueue` contract's logic

Severity: Low Risk.

Description: The current implementation of the `QueryRedeemQueue` logic is lacking some important validations when the demand has already been fully redeemed (`$.demands[demandId - 1].amountIn == 0`).

- The `redeemable` must return `false`.
- The `redeem` function must revert with a custom error like `DemandIdAlreadyRedeemed(demandId);`
- The `resolve` function must revert with a custom error like `DemandIdAlreadyRedeemed(demandId);`
- The `lookup` function must return `0` (no fulfillment available) or maybe even revert with a custom error like `DemandIdAlreadyRedeemed(demandId);`

Recommendation: Kiln should implement the above suggested additional validation checks across the `QueryRedeemQueue` contract's logic.

Kiln: Fixed in commit 086d1980.

Fixed by applying the recommendation — new `DemandIdAlreadyRedeemed` error + `_isFullyRedeemed` sentinel (`amountIn == 0`); `redeemable` / `lookup` return `false` / `0`, `redeem` / `resolve` revert.

Spearbit: Verified the fixes. Fully redeemed demands are now rejected with `DemandIdAlreadyRedeemed` and the view functions return empty values for them.

6.2.11 Partial redemptions uses incorrect rounding in `totalSharesFulfilled` calculations

Severity: Low Risk.

Context: MultiVehicleSharedInternal.sol#L112-L114.

Description: During a partial redemption in the `__tryAutoRedemption` logic, the `totalSharesFulfilled` amount is being calculated with a `Math.Rounding.Floor` rounding.

However, this logic is calculating input shares from output assets. As such, it should use a `Math.Rounding.Ceil` rounding to protect the protocol.

```
function __tryAutoRedemption(
    uint256 extraAssets,
    uint256 totalSupply_,
    uint256 totalAssets_,
    bool skipThresholdCheck
) internal returns (uint256 totalAssetsUsed, uint256 totalSharesFulfilled) {
```

```
    // ...

    // [6] Determine total shares we can fulfill
    if (_availableAssets >= _queueAssetDemand) {
        // ...
    } else {
        // Partial redemption: calculate shares from available assets
        totalSharesFulfilled =
            _previewDeposit(_availableAssets, __assetDecimals, totalSupply_, totalAssets_,
            ↪  Math.Rounding.Floor);
        totalAssetsUsed = _availableAssets;
    }
    // ...
```

Recommendation: Use `Math.Rounding.Ceil` instead.

```
  } else {
      // Partial redemption: calculate shares from available assets
      totalSharesFulfilled =
-         _previewDeposit(_availableAssets, __assetDecimals, totalSupply_, totalAssets_,
↪  Math.Rounding.Floor);
+         _previewDeposit(_availableAssets, __assetDecimals, totalSupply_, totalAssets_,
↪  Math.Rounding.Ceil);
      totalAssetsUsed = _availableAssets;
  }
```

Kiln: Fixed in commit d854caec by applying the recommendation. Partial auto-fulfill `totalSharesFulfilled` switched from `Math.Rounding.Floor` to `Ceil`.

Spearbit: Fixed. The correct rounding is now used.

6.2.12 `MultiVehicle.exitSupplies` can return outdated values

Severity: Low Risk.

Description: The `MultiVehicle.exitSupplies` logic passes to the internal function `__exitSupply` the `totalSupply()` representing the total amount of the `MultiVehicle` shares minted.

This value, depending on the state of the underlying vehicles and the fee manager, could be outdated and not include the new shares that need to be minted to the Fee Manager relative to the performance and management fees.

Recommendation: If the Fee Manager has been configured for the `MultiVehicle`, Kiln must include in the `totalSupply` value passed to `__exitSupply` the amount of shares that would be minted for the performance and management fees.

Kiln: Fixed in commit 8f27c128.

Fixed by applying the recommendation — new `feeAdjustedSupplies()` view returns `(totalSupply, totalAssets)` after pending FeeManager mints; `VehicleManager.exitSupplies` consumes it.

Spearbit: Verified the fixes. `exitSupplies` now uses the fee-adjusted supplies returned by `feeAdjustedSupplies()`.

6.2.13 The `MultiVehicleFacets` threshold `extraAssetsForWithdrawalRequests` should be upper bounded

Severity: Low Risk.

Description: The `extraAssetsForWithdrawalRequests` threshold defined in the `MultiVehicleFacets` contract is used in the `MultiVehicleFacets._createRedeem` logic to add an "extra buffer" of assets to be exited from the underlying vehicles on top of what's already needed to satisfy the current user's redemption needs and the existing `QueryRedeemQueue` unfulfilled demand.

```
    {
        (bool _success, uint256 _totalAssetsToRedeem) = Math.tryAdd(
>>>             _assetsToWithdraw + _queryRedeemQueueAssetsToRedeem,
 ↪  $.thresholds.extraAssetsForWithdrawalRequests
        );
        if (!_success) {
            _totalAssetsToRedeem = type(uint256).max;
        }

        if (_withdrawable < _totalAssetsToRedeem) {
            // Request accounting engine to make assets withdrawable
            // This triggers unallocate() on QueueStrategyEngine, which returns redeem
             ↪  instructions
            // SubQueryEngine then creates redeem queries to sub-vehicles
            $.accountingEngine.requestWithdrawable(_totalAssetsToRedeem);

            // [4] Re-check withdrawable after requestWithdrawable (may have increased if sync
             ↪  redeems)
>>>             _withdrawable = $.accountingEngine.withdrawable();
        }
    }
```

The incorrect configuration of this threshold can lead to two different kinds of problems:

Possible Reverts.

When `MultiVehicleFacets._createRedeem` is executed and `_totalAssetsToRedeem` is set to `type(uint256).max` (or anyway to a very high value) we enter the `_withdrawable < _totalAssetsToRedeem` branch and `$.accountingEngine.requestWithdrawable(_totalAssetsToRedeem);` is executed.

`SectorAccountingEngine.requestWithdrawable` will enter the flow to execute `$.strategyEngine.unallocate(~INF, ...)` and will execute, for each vehicle in the redeem queue, `uint256 _neededRedeemedShares = _currentQueueEntry.vehicle.estimateSingleAssetShares(~INF, false);` which will revert for an overflow exception.

This issue is in general valid for all those cases where the `_totalAssetsToRedeem` is high enough to make the underlying calculations executed by `BaseVehicle._convertToShares` (executed by the `BaseVehicle._estimate`) revert because of **overflow**.

Extra Exits not needed.

As we said, this value is used as an "extra" security buffer of liquidity that will be requested to exit from the underlying vehicles on top of what's really needed.

Every time we deallocate from the redeem queue of vehicles Kiln will.

- Reduce the yield.
- "erode" the total amount of assets owned because of the rounding down that happens on both the vehicles and underlying protocol.

- Reduce the total amount of assets owned because of the redeem fees applied by the underlying vehicles fee managers.

The redeem fees paid to underlying vehicles and protocols are socialized to all the MultiVehicle share holders. Combined with an automatic allocation on deposits, this vector could possibly be leveraged by a malicious party to incur a socialized loss.

Recommendation: Kiln should define a static or configurable "max" value for the `extraAssetsForWithdrawalRequests` threshold to be used as a sanity check upper bound when `MultiVehicle.setThresholds` is executed. It's important to remove the ability to make the redemption process revert and, in general reduce the amount of "extra" liquidity (which could be not needed) to be exited from the underlying vehicles.

Kiln: Fixed in commit 8b95730a.

Fixed by applying the recommendation: `extraAssetsForWithdrawalRequests` is capped at `type(uint128).max` via `setThresholds` reverting `ExtraAssetsForWithdrawalRequestsTooHigh(value, max)`.

Spearbit: The commit 8b95730a solves the possible overflow issue by upper bounding `extraAssetsForWithdrawalRequests` to `type(uint128).max`.

A higher than expected/needed value of `extraAssetsForWithdrawalRequests` does still allow the `MultiVehicle` to withdraw (unallocate) more than needed with all the consequences described in the "Extra Exits not needed" section. We suggest Kiln to document the risk and best practices that the `MULTI_VEHICLE_SET_THRESHOLDS` role should follow when configuring that parameter via `VehicleManager.setThresholds`.

6.2.14 Consider refactoring the `maxDepositable` and `maxRedeemable` functions in `SectorAccountingEngine` to exclude donations

Severity: Low Risk.

Description: Let's assume that the underlying protocol has no deposit/withdraw limit.

- `vehicle.maxDeposit` returns IERC20($.asset).balanceOf(account) → how many ERC20 tokens are owned by the `SubQueryEngine` contract.
- `vehicle.maxRedeem` returns `balanceOf(account)` → how many Vehicle Shares are owned by the `SubQueryEngine` contract.

But those values **could** include "donations" that **cannot** be used when the low-level dispatch operation happens.

Let's take for example `_dispatchRedeem`:

```
uint256 _maxRedeemable =
↪   AssetLib.getSingleAssetValue(vehicle.maxRedeem(address($.subQueryEngine)));
uint256 _redeemableAmount = Math.min(_depositSectorSharesBalance, _maxRedeemable);
```

`maxRedeemable` and `maxDepositable` could right now return a value that is **higher** compared to what it can really redeem/deposit by the dispatch functions.

To make a practical example of where this behavior is problematic we can look at the `_dispatchDeposit` function.

```
uint256 _originalMaxDepositable =
    AssetLib.getSingleAssetValue(vehicle.maxDeposit(address($.subQueryEngine)));

// Cap enforcement
(uint256 _maxDepositable, uint256 _cap, bool _atCap) =
    _computeCapLimitedMaxDepositable(vehicle, _originalMaxDepositable);
```

The value passed to `_computeCapLimitedMaxDepositable` comes from `vehicle.maxDeposit` which could be higher than what it can be really deposited by the `MultiVehicle`. If the protocol has no restriction, it will return the amount of ERC20 tokens owned by the `SubQueryEngine` contract which **could** include **donations**. But donations are "ignored" by the current logic and in reality only what's stored in the Vehicle's sector can be deposited in the vehicle. By using the "donation-affected" balance it's possible that the value used to initialize `_maxDepositable` could result in a higher value than it should.

Recommendation: Kiln should consider refactoring those functions to return the `Math.min` between the `maxDeposit/maxRedeem` of the vehicle and what is accounted for in the vehicle's sector that will be actually used by the redeem/deposit dispatch operations.

After implementing those changes Kiln should start using `maxDepositable` and `maxRedeemable` across the codebase instead of the "plain" `vehicle.maxDeposit` and `vehicle.maxRedeem`.

After implementing the changes above the `_dispatchDeposit` can also be updated to trigger the `DepositLimitedByCap` event with this new conditional logic.

```
  if (
-       _cap > 0 && _maxDepositable < _originalMaxDepositable && _depositableAmount <
↪  _depositSectorAssetBalance
+       _cap > 0 && _maxDepositable < _depositableAmount
  ) {
```

Kiln: Fixed in commit 9ae74f3a.

Fixed by applying the recommendation — internal `_maxDepositable` / `_maxRedeemable` helpers clamp vehicle reads to the actual sector balance so donations to SubQueryEngine cannot inflate dispatched amounts. Public views remain unclamped on purpose (QSE planner needs the raw forward-looking capacity).

Spearbit: Verified the fixes. The internal helpers clamp the vehicle capacity to the actual sector balance.

6.2.15 `_dispatchDeposit` and `_dispatchRedeem` should revert when `DispatchParams` is not empty and returned state is `EMPTY`

Severity: Low Risk.

Description: Both the `_dispatchDeposit` and `_dispatchRedeem` function in `SectorAccountingEngine` could early return or not execute any query depending on the state of the `SectorAccountingEngine` and the dispatch request.

If the `DispatchParams memory params` input parameter is **not** empty and the function does not create and dispatch a query, the transaction should **revert**.

For the moment we can think of the `DispatchParams` parameter as "non empty" when the `minOutput > 0`. We cannot make any assumption relative to the `data` field given that we have no actual practical usage of it for any of the existing vehicles reviewed.

Recommendation: Kiln should revert the `_dispatchDeposit` and `_dispatchRedeem` operations if no query has been created or progressed and the `minOutput` parameter of the `DispatchParams` is greater than zero.

Optional: consider reverting also if the `data` attribute length is greater than zero (non-empty data) depending on the assumption of future use of that parameter.

Kiln: Fixed in commit 6877449e.

Fixed by applying the recommendation — strict `dispatch()` reverts `EmptyStrictDispatch(vehicle, mode, requested, sector)` when both checkpoint blocks complete without producing a query. Non-strict internal callers retain soft-fail by design.

Spearbit: Verified the fixes. Strict dispatches revert with `EmptyStrictDispatch` when no query is produced.

6.2.16 Revert due to zero amount fulfillment may break operations from the MultiVehicle

Severity: Low Risk.

Context: QueryRedeemQueue.sol#L270-L271.

Description: The `QueryRedeemQueue.fulfill()` function reverts when the `amountOutProvided` is zero.

```
function fulfill(uint256 amountInFilled, uint256 amountOutProvided)
    external
    onlyOwner
    returns (uint256 fulfillmentId)
{
    CheckLib.checkValue(amountInFilled);
    CheckLib.checkValue(amountOutProvided); // @audit reverts on zero amount
```

However when the assets per share rate is lower than 1 and the amount of shares to fulfill is 1, the actual correct `amountOutProvided` should be zero. In such conditions, the `QueryRedeemQueue` will revert.

This could lead to a denial of service of any call to `__tryAutoRedemption`, which is used in the `DEPOSIT` and `REDEEM` query `create()` flow and the `feedQueryRedeemQueue` flow.

Note: The `_createRedeem` function is not affected thanks to this check which avoids calling `__tryAutoRedemption` when the amount of assets to fulfill is zero.

Impact: **Low**: `_createDeposit` flow with a zero `minSharesForAutoRedemption` configuration and `feedQueryRedeemQueue` revert.

Likelihood: **Medium**: There must be a `1` share demand in the `QueryRedeemQueue` and an asset per share rate lower than `1`.

Proof of Concept: This proof of concept shows that the issue does not affect the `_createRedeem` flow while it affects the `feedQueryRedeemQueue`.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity >=0.8.33;

import {Query, State} from "src/steam/Query.sol";

import {ErrorLib} from "src/libs/Error.sol";
import {Roles} from "src/libs/Roles.sol";
import {BaseVehicleErrors} from "src/vehicles/base/abstracts/BaseVehicleErrors.sol";
import {IBaseVehicle} from "src/vehicles/base/interfaces/IBaseVehicle.sol";
import {SectorLib} from "src/vehicles/multi/libs/Sector.sol";

import {MultiVehicleTestInternals} from "./MultiVehicleTestInternals.sol";

contract MultiVehicleDustDoSTest is MultiVehicleTestInternals {
```

```
using SectorLib for IBaseVehicle;

/// @dev Reproduces a DoS on `MultiVehicle.feedQueryRedeemQueue()` when:
///      - `redeemQueue.unredeemable()` is non-zero (dust share demand),
///      - but the floor conversion `previewRedeem(unredeemable)` rounds to 0 assets.
///
///      In that state `__tryAutoRedemption()` computes `totalSharesFulfilled > 0` and
↪  `totalAssetsUsed == 0`
///      and then reverts in `QueryRedeemQueue.fulfill()` due to
↪  `CheckLib.checkValue(amountOutProvided)`.
function test_FeedQueryRedeemQueue_DoS_dueToZeroAssetDemand() public {
    IBaseVehicle _vehicleOne = _deploySyncVehicle($asset);

    address _queueOperator = makeAddr("queue_operator");
    address _sectorOperator = makeAddr("sector_operator");
    address _feeder = makeAddr("feeder");
    address _user = makeAddr("user");

    _grant(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION, _queueOperator);
    _grant(Roles.MULTI_VEHICLE_SET_QUEUES, _queueOperator);
    _grant(Roles.MULTI_VEHICLE_MOVE_SHARES, _sectorOperator);
    _grant(Roles.MULTI_VEHICLE_DISPATCH, _sectorOperator);
    _grant(Roles.MULTI_VEHICLE_FEED_QUERY_REDEEM_QUEUE, _feeder);

    vm.startPrank(_queueOperator);
    _authorize_vehicle(_vehicleOne);
    _single_vehicle_deposit_queues(_vehicleOne);
    vm.stopPrank();

    // [1] Create a dust unredeemable demand: redeem 1 wei share while exchange rate is
    ↪  1:1.
    Query memory _dustRedeemQuery;
    vm.startPrank(_user);
    _quick_deposit(_user, 100e18);
    assertEq($accountingEngine.withdrawable(), 0, "sanity: withdrawable should be 0 after
    ↪  allocation");
    _dustRedeemQuery = _quick_redeem(_user, 1, State.PROCESSING);
    vm.stopPrank();

    assertEq($redeemQueue.unredeemable(), 1, "sanity: unredeemable should be 1 wei share");

    // [2] Make assets withdrawable (so feedQueryRedeemQueue will try to create a
    ↪  fulfillment).
    uint256 _allocationShares = $accountingEngine.getSectorBalance(SectorLib.ALLOCATION,
    ↪  _vehicleOne);
    assertGt(_allocationShares, 0, "sanity: allocation shares should be >0");

    vm.startPrank(_sectorOperator);
    $accountingEngine.moveShares(SectorLib.ALLOCATION, _vehicleOne.toSector(),
    ↪  _vehicleOne, _allocationShares);
    $accountingEngine.dispatch(_vehicleOne);
    vm.stopPrank();

    uint256 _withdrawableBeforeLoss = $accountingEngine.withdrawable();
    assertGt(_withdrawableBeforeLoss, 1, "sanity: need >1 wei withdrawable to simulate
    ↪  loss");

    // [3] Simulate a 1-wei loss: withdraw 1 wei to MultiVehicle (not included in
    ↪  totalAssets()).
    vm.prank(address($multiVehicle));
    $accountingEngine.withdraw(1);
```

```
        // This makes the share price slightly < 1, so `previewRedeem(1)` floors to 0.
        assertLt($multiVehicle.totalAssets(), $multiVehicle.totalSupply(), "sanity: rate should
        ↪  be <1");

        // Confirm we're in the problematic state: non-zero share demand but zero converted
        ↪  asset demand.
        (, uint256 _sharesDemand, uint256 _convertedAssetDemand) =
        ↪  $multiVehicle.exitSupplies();
        assertEq(_sharesDemand, 1, "sanity: sharesDemand should be 1 wei share");
        assertEq(_convertedAssetDemand, 0, "sanity: convertedAssetDemand should round to 0");

        // [4] Feeding the queue now reverts on `ErrorLib.ZeroValue()` from
        ↪  QueryRedeemQueue.fulfill(amountOutProvided=0).
        vm.startPrank(_feeder);
        vm.expectRevert(abi.encodeWithSelector(ErrorLib.ZeroValue.selector));
        $multiVehicle.feedQueryRedeemQueue();
        vm.stopPrank();

        // Query remains stuck in PROCESSING since no fulfillment can be created.
        assertEq(uint256($multiVehicle.state(_dustRedeemQuery)), uint256(State.PROCESSING),
        ↪  "dust query should be stuck");
    }

    /// @dev Shows the DoS impact on the user redeem lifecycle:
    ///      once the system is in the "dust demand" state, the user redeem query never becomes
    ↪  unlockable,
    ///      so `state(query)` stays PROCESSING and `unlock()` reverts with InvalidState.
    function test_Redeem_DoS_unlockStuckWhenFulfillmentAssetIsZero() public {
        IBaseVehicle _vehicleOne = _deploySyncVehicle($asset);

        address _queueOperator = makeAddr("queue_operator");
        address _sectorOperator = makeAddr("sector_operator");
        address _feeder = makeAddr("feeder");
        address _user = makeAddr("user");
        address _secondUser = makeAddr("second_user");

        _grant(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION, _queueOperator);
        _grant(Roles.MULTI_VEHICLE_SET_QUEUES, _queueOperator);
        _grant(Roles.MULTI_VEHICLE_MOVE_SHARES, _sectorOperator);
        _grant(Roles.MULTI_VEHICLE_DISPATCH, _sectorOperator);
        _grant(Roles.MULTI_VEHICLE_FEED_QUERY_REDEEM_QUEUE, _feeder);

        vm.startPrank(_queueOperator);
        _authorize_vehicle(_vehicleOne);
        _single_vehicle_deposit_queues(_vehicleOne);
        vm.stopPrank();

        // Create dust redeem demand in queue.
        Query memory _dustRedeemQuery;
        vm.startPrank(_user);
        _quick_deposit(_user, 100e18);
        _dustRedeemQuery = _quick_redeem(_user, 1, State.PROCESSING);
        vm.stopPrank();

        _quick_deposit(_secondUser, 10e18);

        uint256 _demandId = $multiVehicle.demand(_dustRedeemQuery);
        assertGt(_demandId, 0, "sanity: demandId should be non-zero");
        assertEq($redeemQueue.pending(_demandId), 1, "sanity: pending should be 1 wei share");

        // Make assets withdrawable, then simulate a 1-wei loss that makes the share price < 1.
```

```
        uint256 _allocationShares = $accountingEngine.getSectorBalance(SectorLib.ALLOCATION,
        ↪  _vehicleOne);
        vm.startPrank(_sectorOperator);
        $accountingEngine.moveShares(SectorLib.ALLOCATION, _vehicleOne.toSector(),
        ↪  _vehicleOne, _allocationShares);
        $accountingEngine.dispatch(_vehicleOne);
        vm.stopPrank();

        vm.prank(address($multiVehicle));
        $accountingEngine.withdraw(1);

        assertLt($multiVehicle.totalAssets(), $multiVehicle.totalSupply(), "sanity: rate should
        ↪  be <1");
        assertGt($accountingEngine.withdrawable(), 0, "sanity: withdrawable should be >0");

        // Keeper cannot fulfill due to 0-asset fulfill reverting.
        vm.startPrank(_feeder);
        vm.expectRevert(abi.encodeWithSelector(ErrorLib.ZeroValue.selector));
        $multiVehicle.feedQueryRedeemQueue();
        vm.stopPrank();

        // User redeem query remains stuck: it never becomes unlockable, so unlock() reverts.
        assertEq(uint256($multiVehicle.state(_dustRedeemQuery)), uint256(State.PROCESSING),
        ↪  "redeem should be PROCESSING");
        vm.startPrank(_user);
        vm.expectRevert(
            abi.encodeWithSelector(BaseVehicleErrors.InvalidState.selector, State.UNLOCKING,
            ↪  State.PROCESSING)
        );
        $multiVehicle.unlock(_dustRedeemQuery);
        vm.stopPrank();

        assertEq($redeemQueue.redeemable(_demandId), false, "demand should remain
        ↪  unredeemable");

        // A later redeemer is not blocked by the dust demand: `_queryRedeemQueueAssetsToRedeem
        ↪  == 0`
        // makes `_createRedeem` skip queue fulfillment and use withdrawable assets for the new
        ↪  redeem.
        uint256 _secondUserShares = $multiVehicle.balanceOf(_secondUser);
        uint256 _secondUserAssetBalanceBefore = $asset.balanceOf(_secondUser);
        Query memory _secondRedeemQuery = _quick_redeem(_secondUser, _secondUserShares,
        ↪  State.UNLOCKING);

        assertEq(
            uint256($multiVehicle.state(_secondRedeemQuery)),
            uint256(State.UNLOCKING),
            "second redeem should be unlockable"
        );

        vm.prank(_secondUser);
        $multiVehicle.unlock(_secondRedeemQuery);

        assertGt($asset.balanceOf(_secondUser), _secondUserAssetBalanceBefore, "second user
        ↪  should receive assets");
        assertEq(uint256($multiVehicle.state(_secondRedeemQuery)), uint256(State.SETTLED),
        ↪  "second redeem should settle");
        assertEq($redeemQueue.pending(_demandId), 1, "dust demand should still be pending");
        assertEq($redeemQueue.unredeemable(), 1, "dust demand should remain unredeemable");
    }
}
```

Recommendation: Do not execute the `QueryRedeemQueue.fulfill()` function if the total assets used value is zero.

```
  // [7] Early return if nothing to fulfill
- if (totalSharesFulfilled == 0) {
+ if (totalSharesFulfilled == 0 || totalAssetsUsed == 0) {
      return (0, 0);
  }
```

Kiln: Fixed in commit fbec433f.

Fixed by applying the recommendation. `MultiVehicleLib.computeAutoRedemption` guard has been extended to bail when `totalAssetsUsed == 0` in addition to `totalSharesFulfilled == 0`, so dust demands no longer DoS `feedQueryRedeemQueue` / `_createDeposit`.

Spearbit: Fixed. The new `computeAutoRedemption` function implements the recommended `totalAssetsUsed == 0` check.

6.2.17 `subQueryStatus` should return `EXEC` for `PAUSED` queries

Severity: Low Risk.

Context: SubQueryEngine.sol#L373-L374.

Description: The `subqueryStatus` function returns `KeeperLib.JobStatus.POLL` when the query state is `PAUSED` or `PROCESSING`.

However the `PAUSED` state expects the action of a `vehicle.resume()` call. In this state, the keeper should execute a `progressQuery` call.

```
function subqueryStatus(SubQuery calldata subQuery, Query calldata query)
    external
    view
    returns (KeeperLib.JobStatus)
{
    Id _subQueryId = subQuery.toId(address(this));
    SubQueryEngineStorage storage $ = _getStorage();
    if ($.subQueries[_subQueryId] == 0) {
        revert UnknownSubQuery(_subQueryId);
    }
    State _queryState = subQuery.vehicle.state(query);
    if (_queryState == State.PAUSED || _queryState == State.PROCESSING) {
        return KeeperLib.JobStatus.POLL;
    }
    return KeeperLib.JobStatus.EXEC;
}
```

Recommendation: Return `KeeperLib.JobStatus.EXEC` when the query is in `PAUSED` state.

```
  function subqueryStatus(SubQuery calldata subQuery, Query calldata query)
      external
      view
      returns (KeeperLib.JobStatus)
  {
      Id _subQueryId = subQuery.toId(address(this));
      SubQueryEngineStorage storage $ = _getStorage();
      if ($.subQueries[_subQueryId] == 0) {
```

```
            revert UnknownSubQuery(_subQueryId);
        }
        State _queryState = subQuery.vehicle.state(query);
-       if (_queryState == State.PAUSED || _queryState == State.PROCESSING) {
+       if (_queryState == State.PROCESSING) {
            return KeeperLib.JobStatus.POLL;
        }
        return KeeperLib.JobStatus.EXEC;
    }
```

Kiln: Fixed in commit 7dbb30be.

Fixed by applying the recommendation. `POLL` is narrowed to `PROCESSING` only. `PAUSED` falls through to `EXEC` so keepers drive `progressQuery → resume()`.

Spearbit: Fixed. The recommendation has been applied.

6.2.18 The `rebalance` function behavior must be refactored

Severity: Low Risk.

Description: The concept of the rebalance operation is to perform these two actions.

- Redeem `amount` of shares from the vehicle `from`.
- Deposit the resulting amount of assets redeemed from the vehicle `from` into the `to` vehicle.

To be able to define the rebalance operations as successful, the deposit operation **must** at least have been ”started” and not failed instantly.

Given such a premise, below we list all the validations and behaviors that the `SectorAccountingEngine._rebalance` function is missing:

- Revert if `_redeemQueryState != SETTLED`.
- Revert if `_depositQueryState == EMPTY || _depositQueryState == REJECTED`.

Recommendation: Kiln should implement all the suggestions listed above.

Kiln: Fixed in commit 55b75cd3.

Fixed via refactor — `rebalance` removed entirely and replaced by composable `move()` + `dispatch()` with strict-mode reverts (`EmptyStrictDispatch`, `UnauthorizedVehicle`, `ZeroBalance`, `DispatchDepositAmountTooHigh` / `DispatchRedeemAmountTooHigh`, `DepositLimitedByCap`) covering every failure mode the finding cared about.

Spearbit: The `rebalance` function has been removed. External utility contracts will be able to ”emulate” the rebalance by a mix of `move` + manual `dispatch` and reverting if the `dispatch` result has not been executed with a `SETTLED` result.

6.2.19 ”Manual” dispatches could ignore the `minOutput` expressed in the `DispatchParams` requirement

Severity: Low Risk.

Description: For multiple reasons, before the execution of the "root call" that will trigger the `_dispatchDeposit` / `_dispatchRedeem` functions, the vehicle could have an existing balance `> 0` of the token needed for the operation.

When we are in this scenario, dispatches that are triggered by "manual" operations like `rebalance` or `dispatch` could end up ignoring the `minOutput` attribute expressed in the `DispatchParams`. That input attribute is used to initialize `query.outpu` which is used as a validation check when the `BaseVehicle.create` function is executed and would revert the query execution if the amount of received tokens (calculated by `BaseVehicle.estimate`) would be less than the requested one.

```
// Check if the required output matches the estimated output
{
    (Asset[] memory _output, bytes32 _appliedFeesConfigId) = _estimate(
        query.input, query.mode, EstimationType.OUTPUT, _handleFees, false, __totalSupply,
        ↪  __totalAssets
    );
    BaseVehicleStore.getStorage().queries[_qid].feesConfigId = _appliedFeesConfigId;

    AssemblyLib.revertIfBytes(_validateConstraints(query, _output));
}
```

Let's take the `_dispatchDeposit` as an example (the behavior and problem in `_dispatchRedeem` would be the same).

The amount that is deposited in the vehicle is **NOT** bound to the actual amount that has triggered the dispatch (let's say a rebalance or a user's deposit) but is exactly what is in `_getSectorAssetBalance(_vehicleSector, $.asset);`

Let's say that we are from a `rebalance` operation and we originally wanted to deposit `10 USDC` but `_getSectorAssetBalance(_vehicleSector, $.asset);` returns `90 USDC`.

The `_dispatchDeposit` will try to deposit into the vehicle a total of `100 USDC`.

Let's suppose that there is 50% slippage, the "deposit 100 USDC" operation will generate 50 shares and not 100 shares.

If the `rebalance` original operation had a `minOutput = 10 shares` (1:1 with the 10 USDC deposit action), the `_dispatchOperation` will **not** revert because the other 90 USDC (already idling in the sector) has contributed to "satisfy" the `minOutput` that was instead bound to "10 USDC deposit" and not to "100 USDC deposit".

Recommendation: One possible solution would be to add a `amount` attribute `DispatchParams` that would **cap** the max amount of tokens deposited/redeemed by the underlying dispatch operation to such value.

It's important to note that passing `DispatchParams.amount = INFNITE` could enable the scenario we are trying to prevent with the suggested changes. The Asset Manager must be warned and aware of that.

Kiln: Fixed in commit 55b75cd3 (PR 453, `DispatchParams.amount` cap).

Fixed by applying the recommendation: the `DispatchParams.amount` cap has been added. The `_createQuery` → `query.input[0].value` → `vehicle.estimate` path isolates the slippage check from any stale sector balance, so a `minOutput` cannot be silently satisfied by pre-existing sector contents. Operators wanting the "use entire sector balance" semantic opt in with `type(uint256).max`.

Spearbit: Partially resolved. The core finding is addressed by the `DispatchParams.amount` cap. Kiln agreed that prior shares must be taken into account when checking the per-vehicle targets and submitted a follow-up fix after the engagement (PR 446, commit 0e1e045) that has not been reviewed by Spearbit; it is listed in the appendix (see the "Focus point document") as requiring further verification, together with confirming that the `_createQuery`

→ `query.input[0].value` → `vehicle.estimate` slippage path can no longer be satisfied by stale `_getSectorAssetBalance` contents, especially when operators opt into `type(uint256).max`.

6.2.20 `lookup()` can loop indefinitely when no fulfillment matches

Severity: Low Risk.

Context: QueryRedeemQueue.sol#L590-L601.

Description: `QueryRedeemQueue._lookupFulfillment()` performs a binary search inside an unbounded `while (true)` loop. When the searched position is not covered by any fulfillment in the selected range, the search bounds may stop progressing.

```
function _lookupFulfillment(uint256 startPosition, uint256 startFulfillmentId, uint256
↪  endFulfillmentId)
    internal
    view
    returns (uint256)
{
    // ...

    while (true) {
        uint256 _selectorIndex = (_startIndex + _endIndex) / 2;
        Fulfillment memory _selectorFulfillment = $.fulfillments[_selectorIndex];
        if (_isMatchingFulfillment(startPosition, _selectorFulfillment)) {
            return _selectorIndex + 1;
        }

        // Checks if a position is after a fulfillment's range (for binary search). // @audit
        ↪  if there is no matching fulfillment, this will never break the loop
        if (startPosition >= _selectorFulfillment.position +
        ↪  _selectorFulfillment.filledAmountIn) {
            _startIndex = _selectorIndex;
        } else {
            _endIndex = _selectorIndex;
        }
    }
    return 0;
}
```

This can be reached through `lookup()`, which calls `_lookupFulfillment()` whenever at least one fulfillment exists. The interface documents that `lookup()` should return `0` if no matching fulfillment exists, but the current helper only reaches its trailing `return 0` after an infinite loop, making that return path dead code. A fully redeemed demand that is looked up again can therefore consume all gas instead of returning a clear no-match result.

Proof of Concept:

```
pragma solidity >=0.8.33;

import {Test, console} from "@forge-std/Test.sol";
import {ErrorLib} from "src/libs/Error.sol";
import {QueryRedeemQueue} from "src/vehicles/multi/QueryRedeemQueue.sol";
import {ERC20} from "@openzeppelin-contracts/token/ERC20/ERC20.sol";
import {IERC20} from "@openzeppelin-contracts/token/ERC20/IERC20.sol";
import {Math} from "@openzeppelin-contracts/utils/math/Math.sol";
import {IQueryRedeemQueue} from "src/vehicles/multi/interfaces/IQueryRedeemQueue.sol";

contract MockERC20 is ERC20 {
```

```
    constructor(string memory name_, string memory symbol_) ERC20(name_, symbol_) {}

    function mint(address account, uint256 value) external {
        _mint(account, value);
    }
}

contract _SQRedeemTest is Test {

    QueryRedeemQueue qr;
    MockERC20 t1 = new MockERC20("1", "1");
    MockERC20 t2 = new MockERC20("2", "2");

    function setUp() public {
        qr = new QueryRedeemQueue();
        qr.initialize(address(this), t1, t2);

        t1.mint(address(this), 1000e18);
        t2.mint(address(this), 1000e18);

        t1.approve(address(qr), type(uint256).max);
        t2.approve(address(qr), type(uint256).max);
    }

    function testLookupOOG() public {
        uint256 d1 = qr.demand(15, 15);
        qr.demand(100, 100);
        qr.demand(100, 100);
        qr.demand(100, 100);
        uint256 f1 = qr.fulfill(100, 100);
        qr.fulfill(100, 100);
        qr.fulfill(100, 100);

        console.log('=== REDEEM d1');
        qr.redeem(d1);

        qr.lookup(1);
    }


}
```

Recommendation: Consider adding a loop breaker when `_startIndex == _endIndex`.

Also, add regression tests on this function to ensure that this issue is resolved.

Kiln: Fixed in commit 9722d157 and commit 5f19d28.

Fixed by applying the recommendation. `_lookupFulfillment` self-protects with an early `return 0` once `startPosition >= _endFulfillment.position + filledAmountIn`. Regression test has been added.

Spearbit: Fixed. The new logic will return `0` when no fulfillment matches the demand.

6.2.21 `QueueStrategyEngine.allocate` should not use the same `Target` to both validate the max exposure and delta target

Severity: Low Risk.

Description: The `Target.constrainBy` is used by the `QueueStrategyEngine` during the `allocate` function.

The ”picked” target (returned by the `Target.constrainBy` function) will then be used to execute.

```
if (_effectiveTarget.isAtTarget(_vehicleSharesHoldings)) {
    continue;
}
```

This logic will **skip** the vehicle if `_vehicleSharesHoldings >= effectiveTargetValue` where `effectiveTargetValue = t.value >= t.threshold ? t.value - t.threshold : 0`.

As you can see the `Target.effectiveValue` **uses** the `threshold` attribute from the target, but the `Target.min` function that selects the min `Target` **does ignore** it.

The **wrong** target might be selected by `constrainBy` because of that.

Let's make an example.

- `t1 = Target({ value: 100, threshold: 10 })`
- `t2 = Target({ value: 95, threshold: 0 })`

Executing `Target.min(t1, t2)` with the current implementation would return `t2` because `t2.value < t1.value` but in reality if we consider **where** the target is used and **how** the threshold is used in that context, the **correct** target to be returned by `Target.min` should be `t1` because `t1-threshold < t2-threshold`.

Because of this it's possible that `QueueStrategyEngine.allocate` **will not** skip a vehicle even if the current's vehicle holdings are **already** above the target's threshold.

Recommendation: We suggest Kiln to follow this logic to pick the correct `Target` depending on which is the context where it will be used and how it will be used:

- Use the **most conservative** target when `isAtTarget(_vehicleSharesHoldings)` is executed (the one with the lowest `value-threshold`).
- Use the target with the lowest `value` when `_deltaSharesToTarget` is calculated.

Kiln: Acknowledged — intentional behavior.

`Target.constrainBy` picks the lower-`value` target because `value` is the binding constraint and `threshold` is an operator-defined tolerance applied on top by `isAtTarget`. Operators wanting a stricter skip set the threshold directly on whichever target they consider authoritative.

Spearbit: Acknowledged.

6.2.22 `_dispatchDeposit()` and `_dispatchRedeem()` Early Returns Leave VEHICLE Sector Assets Stranded

Severity: Low Risk.

Context: SectorAccountingEngine.sol#L748.

Description: In `SectorAccountingEngine._dispatchDeposit()`, there are several paths that skip creating a deposit sub-query while leaving `asset()` tokens already staged in the vehicle sector: when the vehicle is not authorized, when the vehicle is at cap, and when `maxDeposit` returns 0. In all cases `asset()` tokens were already moved from DEPOSIT into the vehicle sector by the caller before `_dispatchDeposit` was invoked. These funds are not lost — they will be picked up on the next call to `deposit()` or `dispatch()`. However if the allocation

strategy changes before that happens, the stranded `asset()` tokens may not be automatically re-routed and an admin would need to manually call `moveAssets()` to release them.

`_dispatchRedeem()` has a similar issue when `maxRedeem` returns 0 — shares already moved from ALLOCATION into the vehicle sector are left stranded with no redeem sub-query created. This case is significantly less probable than the deposit case — it requires the sub-vehicle to explicitly block redemptions (e.g. paused or in a cooldown period).

Recommendation: Emit an event on all early return paths so that admins can detect and respond to stranded assets off-chain without relying on manual inspection.

Kiln: Fixed in commit 55b75cd3.

Fixed by applying the recommendation — `LimitedDeposit` / `LimitedRedeem` events fire on every reachable early-return path in `_dispatchDeposit` / `_dispatchRedeem` (at-cap, `maxDeposit` / `maxRedeem` returning 0), with `actualAmount=0` notation when capacity is fully consumed. The unauthorized non-strict branch is pre-filtered by `QSE.allocate` / `unallocate` and so unreachable in practice. The operator how-to consolidates the recovery patterns (see the response to the finding on the silent-fail behavior of `_dispatchDeposit` and `_dispatchRedeem`).

Spearbit: Partially resolved. The `LimitedDeposit` / `LimitedRedeem` events now cover the at-cap and `maxDeposit` / `maxRedeem` returning 0 early-return paths, but the `!_isAuthorizedVehicle()` branch with `strict = false` still appears to leave sector `asset()` tokens stranded without an event, the `_resolvedAmount = 0` path needs its reachability assessed, and `_dispatchDeposit` should arguably skip the `_allocations` loop iteration when `_allocation.asset.value` is 0 instead of activating the vehicle for a no-op. Kiln submitted a follow-up fix after the engagement (PR 446, commit 8e3e308) that has not been reviewed by Spearbit; these items are listed in the appendix (see the "Focus point document").

6.2.23 User slippage protection is ineffective on MultiVehicle redeem queries

Severity: Low Risk.

Description: The `MultiVehicle` implements an asynchronous redeem mechanism due to liquidity constraints. As it allocates funds to underlying vehicles, it may not always have enough liquidity to cover redeem queries. For this, the `QueryRedeemQueue` contract exists. When not enough liquidity is available to cover for a user redeem, a demand is added in the FIFO queue. A demand has two main parameter: the amount of shares to redeem `amountIn` and a maximum amount of assets `maxAmountOut`.

```
/// @notice Creates a new redemption demand in the queue.
/// @dev Transfers shares from the owner (multivehicle) to this contract.
/// @param amountIn The amount of shares to redeem.
/// @param maxAmountOut The maximum amount of base assets expected in return.
/// @return demandId The demand ID.
function demand(uint256 amountIn, uint256 maxAmountOut) external onlyOwner returns (uint256
↪  demandId) {
```

Even though the user created their redeem query with an output constraints checked in the `BaseVehicle` to ensure they receive a minimum amount of assets, the corresponding demand may be fulfilled in the future with way less assets than this `maxAmountOut`. This makes the slippage protection on redeem query completely ineffective.

```
/// @inheritdoc IVehicle
function create(Query calldata query)
    external
    override
    nonReentrant
```

```
    steamOperation(query, State.EMPTY)
    returns (State newState)
{
    // ...

    // Check if the required output matches the estimated output
    {
        (Asset[] memory _output, bytes32 _appliedFeesConfigId) = _estimate(
            query.input, query.mode, EstimationType.OUTPUT, _handleFees, false, __totalSupply,
            ↪  __totalAssets
        );
        BaseVehicleStore.getStorage().queries[_qid].feesConfigId = _appliedFeesConfigId;

        AssemblyLib.revertIfBytes(_validateConstraints(query, _output)); // @audit this may not
        ↪  be respected at `unlock()` time
    }
```

Recommendation: It should at least be documented that redeem queries are not slippage protected in the MultiVehicle.

Otherwise, the slippage protection mechanism could be reviewed to give back to the user their shares if the constraints is not satisfied. Note that this could add a lot of complexity.

Kiln: Fixed in commit 46c869c1 by applying the minimum-bar recommendation. The caveat is documented at all four surfaces: `VEHICLE.md` table callout, QRQ section bullet, `QueryRedeemQueue.demand()` natspec, `MultiVehicleFacets._createRedeem` natspec.

Spearbit: Fixed through documentation. The behavior is acknowledged by Kiln and is documented. Users should be aware of it.

6.2.24 `unallocate()` `complete` Return Value Ignored Causing Silent Liveness Failure

Severity: Low Risk.

Context: SectorAccountingEngine.sol#L448.

Description: In `SectorAccountingEngine.requestWithdrawable()`, the `complete` return value of `strategyEngine.unallocate()` is silently discarded. When `unallocate()` returns `complete=false` (e.g. due to a misconfigured redeem queue, low caps, or all vehicles in manual mode), full sub-queries will not be created, but rather only partial or no sub-queries at all. However `_createRedeem` proceeds to call `demand()` and transitions the query to PROCESSING — creating a stuck state where Alice's query has a valid `demandId` but no sub-queries in flight to generate liquidity. `feedQueryRedeemQueue()` early returns since `_withdrawableAssets = 0`, `progressQuery()` has nothing to call, and `unlock()` returns PROCESSING indefinitely. No error or event is emitted to signal the failure.

Recommendation: Emit an event when `complete=false` so the gated role responsible for `moveShares` can detect and respond off-chain by:

1. Debugging why `unallocate()` did not complete (misconfigured redeem queue, low caps, etc.).
2. Fixing the configuration appropriately.
3. Calling `moveShares(ALLOCATION, vehicle.toSector(), vehicle, deltaAmount)` for the amount that was not unallocated — the delta should be taken from the event — using a vehicle that can be withdrawn from.
4. Calling `dispatch(vehicle)` to create the redeem sub-query.

Kiln: Fixed in commit 53539e8 (residual events for partial deposit + withdraw shortfall (27, 52)).

Fixed by applying the recommendation — `RequestWithdrawableShortfall(requestedAmount, plannedUnallocation, manualShortfall)` event added; `QSE.unallocate` signature extended with `remainingAssets` so the engine no longer discards the incomplete-plan signal. The operator how-to (`operate-multivehicle.md` "Handle stranded sector balances", added in the same commit) walks through the recovery flow.

Spearbit: Fixed by implementing the reviewer's recommendation.

6.2.25 `_createRedeem` could return less assets than the expected amount

Severity: Low Risk.

Description:

Scenario 1: For the sake of simplicity let's assume that no fee manager (in both the MV and the underlying vehicles) has been configured. The user has requested to redeem `sharesToRedeem` and the amount of assets received is exactly equal to.

`uint256 _assetsToWithdraw = _previewRedeem(sharesToRedeem, ..., Math.Rounding.Floor)`

This value matches what the user has estimated to get before creating the MV query by setting `query.output = MV.estimate(sharesToRedeem, Mode.REDEEM, EstimationType.OUTPUT)`.

Let's assume that `_withdrawable > 0` AND `_withdrawable < _assetsToWithdraw`. We are in the scenario where we have SOME ASSETS available to be "instantly" withdrawn but they are not enough to fully cover the user's request. This means that we need to partially request the redeem of SOME Vehicle shares (to fully satisfy the user's request).

Let's assume that we are in the edge case scenario where `_withdrawable ~= _assetsToWithdraw` (lower by 1 wei). Because of the **ROUNDING UP**, it's possible that `uint256 _immediateRedeemShares = _previewDeposit(_toWithdraw, ..., Math.Rounding.Ceil);` will result in `_immediateRedeemShares > 0` AND `_immediateRedeemShares == sharesToRedeem`.

In this scenario `_sharesLeftToRedeem == 0` and we would skip the whole `if (_sharesLeftToRedeem > 0) {` conditional branch executing the last part of the logic.

```
// [9] Full synchronous redemption - store unlock amount and burn shares
$.values[qid].amountToUnlock = _toWithdraw;
_burn(address(this), sharesToRedeem);

// [10] Transition to UNLOCKING state
return _transition(qid, State.UNLOCKING);
```

Which is instantly **UNLOCKING** the user's redeem operation.

We have two issues in this scenario:

1) The user is at the end getting LESS ASSETS than the one expected and promised while burning the same amount of shares.
   - `sharesToRedeem` shares are burned (the one from the original user's request).
   - `_toWithdraw` assets are given to the user that are **LESS** than the calculated and expected one `_assetsToWithdraw`.
2) The `_createRedeem` in this scenario is not respecting the invariant held by the BaseVehicle when `AssemblyLib.revertIfBytes(_validateConstraints(query, _output));` has been executed during the `BaseVehi`

`cle.create` function. The user had configured the `q.output = BaseVehicle.estimate(sharesToRedeem, Mode.REDEEM, EstimationType.OUTPUT)`.

Scenario 2: Let's assume that we are almost in the same scenario as the one above but.

`uint256 _immediateRedeemShares = _previewDeposit(_toWithdraw, ..., Math.Rounding.Ceil)` **ROUNDS UP** to be 1 wei lower than `sharesToRedeem` In this case we have `_sharesLeftToRedeem == 1`.

We enter the `if (_sharesLeftToRedeem > 0) {` conditional branch but because of the ROUNDING DOWN of `uint256 _assetsLeftToRedeem = _previewRedeem( _sharesLeftToRedeem, ..., Math.Rounding.Floor)` we end up with `_assetsLeftToRedeem == 0`.

We skip the code in the `if (_assetsLeftToRedeem > 0) {` conditional branch and we execute the same final block of "Scenario 1".

The result will be the same with the same problem described above.

Recommendation: Kiln has two potential options to evaluate, brainstorm about and properly validate with proper testing.

1) Do not perform an "instant" withdraw (going directly into the UNLOCKING phase) in those cases but instead create a full demand for `$.redeemQueue.demand(sharesToRedeem, _assetsToWithdraw)` and transition to the `PROCESSING` state. The user will need to wait but they at least they will get the full deserved amount.
2) Properly document those scenarios as edge cases and disclose that users could get less than what has been estimated even if the `BaseVehicle` has successfully passed the invariant `AssemblyLib.revertIfBytes(_validateConstraints(query, _output));`

Kiln: Fixed in commit cc079fb7 (post-create upper-bound output check in `_createRedeem`).

Fixed by applying a stricter variant — `_checkRedeemOutput` helper added; all three terminal branches of `_createRedeem` (full-sync, partial-sync, full-async) compute a deterministic upper-bound payout and revert `InvalidEstimation` when it falls below the user's `query.output[0].value` floor. Covers both scenarios in the finding.

Spearbit: With the commit cc079fb7 Kiln is validating the amount the user is going to instantly unlock or partially unlock (+ estimate future unlock) against the expected value.

6.2.26 Functions that could change total assets/shares must be routed via `MultiVehicle` and wrapped with Fee Manager fee logic

Severity: Low Risk.

Description: There are some actions that can be executed outside the `MultiVehicle` by authed roles that can change MultiVehicle's total assets and impact the exchange rate. Every action that has that effect should be "wrapped" by the Fee Manager's fee logic available only in the `MultiVehicle`'s contract.

```
(uint256 __totalSupply, uint256 __totalAssets) =
    _handleOngoingFees(Id.wrap(bytes32(0)), BaseVehicleStore.getStorage().feeManager, true);

// TODO: execute the logic

_updatePostFeeCheckpoints();
```

In the current implementation of the codebase we have identified these functions:

- `SectorAccountingEngine.deposit`: users with the `MULTI_VEHICLE_DEPOSIT` role can directly inject liquidity and increase the total assets. The caller can decide to allocate it or not via the `allocate` input parameter.
- `SectorAccountingEngine.rebalance`: users with the `MULTI_VEHICLE_REBALANCE` role can indirectly influence the total assets of the `MultiVehicle` by redeeming from a vehicle and depositing into another. The total assets can be influenced by the underlying vehicles fee management and the underlying protocols (used by those vehicles) fee manager.
- `SectorAccountingEngine.dispatch`: users with the `MULTI_VEHICLE_DISPATCH` role can influence the total assets of the `MultiVehicle` by dispatching assets/shares already allocated for deposits/redeem operations (in the past) because of the underlying vehicle fee management.
- `SubQueryEngne.progressQuery`: users with the `MULTI_VEHICLE_PROGRESS_QUERY` role can influence the total assets of the `MultiVehicle` by progressing "async" queries which could return (when unlocked/recovered) an output of assets lower/greater than the one previously estimated in the ephemeral accounting.

Recommendation: All the above listed actions (and future one) must be directly executed by the `MultiVehicle` and wrapped with the existing Fee Manager's fee management logic.

Kiln: Acknowledged — intentional behavior.

The four privileged entrypoints (`SectorAccountingEngine.deposit` / `move` / `dispatch` , `SubQueryEngine.progressQuery`) are part of the investment-strategy surface — they shift `totalAssets()` because they allocate liquidity, rebalance positions, or settle async queries. Earnings and losses from those moves are investment performance, not donations, and should accrue performance fees. Wrapping them in `startOngoingFeeHandling` / `finishOngoingFeeHandling` would snapshot the post-op state into the FeeManager baseline, effectively treating each privileged delta as donation-like and zeroing the performance fee on it. We leave them unwrapped so the next fee-aware call (a user deposit / redeem, `feedQueryRedeemQueue` , or `retrieveQueryRedeemQueueAssets`) captures the strategy's realised PnL.

Spearbit: Acknowledged.

6.2.27 Vehicle exposure caps can be underestimated because `sharesBeforeCreates` are excluded

Severity: Low Risk.

Context: QueueStrategyEngine.sol#L216-L224, SectorAccountingEngine.sol#L1193-L1195.

Description: `SectorAccountingEngine._getActiveHoldings()` excludes `sharesBeforeCreates` , which are shares held in the vehicle sector before a redeem query is created. They actually represent shares allocated in the underlying vehicle.

The current assumption is that these shares are "on their way out" and therefore should not count toward active vehicle holdings. However, those shares are still owned by the MultiVehicle and still represent exposure to the underlying vehicle. Shares can be accumulated in this sector through two ways:

- Asset Manager has moved those shares explicitly into the vehicle sector via `moveShares` without a `_dispatchRedeem` .
- Shares were automatically (e.g. unallocation) moved into the vehicle sector via `requestWithdrawable` but the `_dispatchRedeem` has "silently" failed (e.g. redeemable amount is zero).

The same active holdings logic is used for cap-sensitive decisions such as allocation and rebalancing. Because `sharesBeforeCreates` are excluded, the system may underestimate the real exposure to a vehicle and allow additional funds to be allocated or rebalanced into it.

Example:

- Vehicle target/cap is 100 shares.
- MultiVehicle owns 100 shares.
- 40 shares are staged for redemption and moved out of the allocation sector.
- Active holdings are computed as 60.
- The system allows another 40 shares to be allocated to reach the target again.
- If the pending redeem is recovered or does not complete, the 40 shares return.
- The MultiVehicle now holds 140 shares, exceeding the intended cap.

This weakens the protection provided by allocation and rebalance caps, especially because cap checks should conservatively overestimate exposure rather than underestimate it.

Recommendation: Kiln should think about if `sharesBeforeCreates` should be included in the active holdings value used for allocation caps. Unallocations should probably not account for these shares.

The cap logic should treat shares held in the vehicle sector as active exposure until they are actually redeemed and no longer owned by the MultiVehicle.

Kiln: Fixed in commit be8be991.

Fixed by applying the recommendation. `_vehicleEstimatedShares` (cap-side view) now includes `sharesBeforeCreates` so staged-redeem shares cannot create fake cap headroom. `vehicleSettledShares` (used by `QSE.unallocate`) intentionally still excludes them — the redeem planner needs the deliverable balance, not the exposure (matches the finding's "unallocations should probably not account for these shares").

Spearbit: Fixed. The vehicle exposure is now correctly estimated to protect the protocol.

6.2.28 Stale `totalAssets_` in `_createRedeem()` After `requestWithdrawable`

Severity: Low Risk.

Context: MultiVehicleFacets.sol#L394.

Description: In `_createRedeem()`, `totalSupply_` and `totalAssets_` are captured at the start of `BaseVehicle.create()`. When `requestWithdrawable()` synchronously settles sub-vehicle positions, any withdrawal fees charged by those sub-vehicles reduce `accountingEngine.totalAssets()` with no corresponding burn of MultiVehicle shares. This breaks the `totalAssets / totalSupply` ratio: `totalAssets_` becomes overstated while `totalSupply_` remains correct, causing `_assetsLeftToRedeem` recorded in the queue demand to be slightly inflated.

`__tryAutoRedemption()` + the subsequent `_burn` do not require a re-read: both reduce numerator and denominator by amounts derived from the same ratio, preserving it to within 1 wei of floor-rounding dust.

Recommendation: Re-read `totalAssets_` (not `totalSupply_`) after `requestWithdrawable()`:

```
if (_withdrawable < _totalAssetsToRedeem) {
    $.accountingEngine.requestWithdrawable(_totalAssetsToRedeem);
    _withdrawable = $.accountingEngine.withdrawable();
    // Sub-vehicle withdrawal fees reduce totalAssets() without burning shares — re-read to
    ↪  correct the ratio.
    totalAssets_ = $.accountingEngine.totalAssets();
}
```

Kiln: Fixed in commit a19e548d.

Fixed by applying the recommendation — `_createRedeem` refreshes `totalAssets_` after `requestWithdrawable()` and recomputes queue demands via `exitSupplies` before the share-split calculation, so any sub-vehicle redeem fees crystallized during the dispatch path are reflected.

Spearbit: Fixed by implementing the reviewer's recommendation.

6.2.29 Unauthorizing an async vehicle with a PROCESSING query strands its sector funds

Severity: Low Risk.

Context: SubQueryEngine.sol#L469-L480.

Summary: When an async sub-vehicle has a query in `PROCESSING` state and is then unauthorized via the `VehicleRegistry`, the eventual settlement still drives `SubQueryEngine` to call back into the accounting engine and credit the unauthorized vehicle's sector. `_addToSector` trusts `SubQueryEngine` and writes the balance, but downstream `dispatch` for that vehicle now early-returns on the `_isAuthorizedVehicle` check, so the funds sit in the vehicle's sector with no automatic path to move them out.

Impact: Assets accumulate in the unauthorized vehicle's sector after a PROCESSING query settles. Because `dispatch` short-circuits for unauthorized vehicles, no automatic flow (allocate, unallocate, dispatch) will sweep them. Recovery requires the operator to **manually** move assets/shares out of the stranded sector via the privileged `moveAssets` / `moveShares` paths, and there is no on-chain enforcement that this happens, so an inattentive admin can leave funds idle (and unaccounted-for in some downstream views) after a deauth.

Recommendation: Either (a) block `unauthorize` while the vehicle has any non-zero ephemeral accounting (PROCESSING queries) or non-empty vehicle sector, forcing the operator to drain first; or (b) document the post-unauth cleanup requirement explicitly and provide a single helper (e.g., `drainUnauthorizedVehicle(vehicle)`) that performs the manual moves atomically, so admins can't forget a step.

Kiln: Acknowledged — operator responsibility.

`VehicleManager.unauthorize` does not block on `accountingEngine.isVehicleActive(vehicle)`. The operator holding `MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION` is responsible for draining any in-flight queries and emptying the vehicle sector before unauthorising; if a settled query later credits an orphaned sector, the privileged `move` paths (gated by `MULTI_VEHICLE_MOVE`) recover the balance. No on-chain guard is added — it would foreclose emergency-deauth of a sub-vehicle that is structurally unable to settle its queries, where forcing a ”drain first” precondition would lock the funds permanently.

Spearbit: This is accepted by Kiln that in such cases the funds should be manually moved by operators.

6.2.30 Residual `maxAmountOut` should be reduced proportionally, not by the amount paid

Severity: Low Risk.

Context: QueryRedeemQueue.sol#L523.

Summary: When a demand is partially filled, `maxAmountOut` is decremented by the actual asset amount paid for the overlapping portion instead of by the proportional slice of the original cap. The residual demand inherits whatever rate the first fulfillment delivered, so its per-share cap drifts and subsequent fulfillments pay it differently from sibling demands that were submitted under the same original expectation.

Impact: The same fulfillment can pay two demands at the same per-share rate but produce different outcomes, depending on whether one of them was previously partially filled at a worse rate. Outcomes also depend on fulfillment ordering (which the user does not control): the same set of fulfillments can leave a user whole if the unfavorable one arrives first, or short-paid if the favorable one arrives first.

Proof of Concept: Alice: 100 shares, `maxAmountOut = 100` (cap 1.0 per share). Bob: 50 shares, `maxAmountOut = 50` (cap 1.0 per share).

1. Fulfillment 1 (0.8 per share) pays Alice 40 for 50 shares. Her residual cap drifts to `60 / 50 = 1.2`.
2. Fulfillment 2 (1.2 per share, 100 shares total) pays Alice 60 for her remaining 50 shares (at the drifted cap), and Bob only 50 for his 50 shares (capped at his original 1.0). Alice ends with 100, Bob with 50, on identical per-share expectations.

Reverse order: if Fulfillment 2 arrives first, Alice is paid 50 at her original cap, her residual cap stays at 1.0, Fulfillment 1 then pays her 40 for 50 shares. Alice ends with 90 instead of 100, even though the combined fulfillments averaged above 1.0 per share.

Recommendation: Change:

```
$demand.maxAmountOut -= _minRedeemableAmount;
```

to:

```
$demand.maxAmountOut -= _demandOverlappingAssetAmount;
```

The decrement should consume the proportional slice of the demand's original cap, not the actual asset amount paid. This keeps the per-share cap fixed across partial fills, so a given fulfillment treats every demand at its own per-share expectation regardless of fulfillment ordering, and any excess from favorable fulfillments correctly accrues to `retrievable`.

Kiln: Acknowledged — intentional behavior.

`maxAmountOut` is by design a maximum on the total assets a demand can receive, not a per-share commitment. Decrementing it by the amount actually paid (not by a proportional slice of the original cap) is what lets a demand recover via later, more favorable fulfillments — the residual cap is preserved for them to draw from. Reducing the cap proportionally would lock in the worst rate seen and forbid that recovery, which would be the real unfairness given that `maxAmountOut` is the upper bound the user signed up for, not the rate they accepted. The order-dependent outcomes the audit walks through are a consequence of operator-controlled fulfillment scheduling, not a deviation from the cap-as-maximum semantic.

Spearbit: Acknowledged.

6.3 Informational

6.3.1 Dead code

Severity: Informational.

Context: SubQuery.sol#L48-L53.

Description: The codebase contains several instances of unused code like custom errors, library functions, and role constants that are defined but never referenced anywhere in the project.

- ☒ `src/vehicles/multi/libs/SubQuery.sol:52` : `getSector` function unwraps and re-wraps a given `Sector`. This does not contain specific logic.

- ☒ `src/vehicles/multi/libs/Sector.sol:107-119` : `isQuerySector` and `isStaticSector` are executing `bytes1(sector) & 0xFF`. However, masking 1 byte with `0xFF` does not change the output. The `& 0xFF` code can be removed.
- ☒ `src/vehicles/multi/libs/Target.sol:81` : The `remainingCapacity` function is implemented but never used throughout the codebase, even though it could be used at multiple locations such as `Sector AccountingEngine.sol:1231`.
- ☒ `src/vehicles/multi/SubQueryEngine.sol:604-606` : Values are overwritten with zeros before being deleted through the `$assets.pop();` call. Overwriting with zeros is not required as this is done by the compiler through the `pop()` operation. Also, `delete $assets` could be done outside the loop.
- ☒ `src/vehicles/multi/QueryRedeemQueue.sol:142-144` : `UnsupportedAsset` error is never used.

Recommendation: Dead code should be removed from the codebase where possible. Removing unused declarations reduces code size, lowers the risk of confusion for developers and auditors, and eliminates potential inconsistencies.

Kiln: Fixed in commit 93777938.

Fixed by applying the recommendation. Note: `TargetLib.remainingCapacity` and `QueryRedeemQueue.UnsupportedAsset` were later re-introduced by commit 76584db4 and commit d4a6bf3f respectively — with active uses, so they are no longer dead.

Spearbit: Fixed.

6.3.2 Ineffective logic in `constrainBy`

Severity: Informational.

Context: Target.sol#L112-L119.

Description: The `constrainBy` logic implements two calls to `isUnlimited` before returning the `min(t, cap)` value.

However, these two `isUnlimited` calls are not required as they do not modify the output logic when the same values are passed to `min(t, cap)`. This is because an input is recognized as unlimited when it is `type(uint256).max` value.

```
function isUnlimited(Target memory t) internal pure returns (bool) {
    return t.value == type(uint256).max;
}

function constrainBy(Target memory t, Target memory cap) internal pure returns (Target memory)
↪   {
    if (isUnlimited(t)) {
        return cap; // Target unlimited → cap is the limit
    }
    if (isUnlimited(cap)) {
        return t; // Cap unlimited → target is the limit
    }
    return min(t, cap); // Both limited → pick lower
}
```

Recommendation: Consider removing the `isUnlimited` calls.

```
  function constrainBy(Target memory t, Target memory cap) internal pure returns (Target
  ↪  memory) {
-     if (isUnlimited(t)) {
-         return cap; // Target unlimited → cap is the limit
-     }
-     if (isUnlimited(cap)) {
-         return t; // Cap unlimited → target is the limit
-     }
      return min(t, cap); // Both limited → pick lower
  }
```

Kiln: Fixed in commit 6d997990.

Fixed by applying the recommendation, `constrainBy` collapsed to a single `min(t, cap)` call.

Spearbit: Fixed. Recommendation has been applied.

6.3.3 Bulk Informational Issues

Severity: Informational.

Description:

1) [ ] Consider renaming the `Thresholds.minSharesForAutoRedemption` attribute `minSharesForAutoFulfill` and the `__tryAutoRedemption` function to `__tryAutoFulfill` given the usage of those attribute/functions.
2) [ ] Consider creating a util function in `TargetLib` that will "generate" a new `Target` struct object that validates and enforces specific sanity checks. For example, when the `Target.value` is "unlimited" (`type(uint256).max`), the `threshold` should be enforced to **zero**.
3) [ ] Consider re-evaluating the events emitted during the contract's initialization phase. For example the `VehicleRegistry` , unlike other contracts, does not emit any event when the `initialize` function is executed. Other contracts instead, like (for example) the `SubQueryEngine` emit events like `MultiVehicleEvents.MultiVehicleInitialized` which could be seen as misleading given that the MultiVehicle contract itself is **not** initialized.
4) [ ] Remove the `owner` state variable from the `QueryRedeemQueue` and `SectorAccountingEngine` contracts. Those contracts accept both an owner and the multivehicle itself but in reality the owner is indeed the multivehicle (based on the current logic and behavior).
5) [ ] Review the codebase and look for code with "repeated" patterns and consider refactoring them to improve the code's readability. For example in the `SectorAccountingEngine._moveFromSector` the `IERC20(_currentAsset.asset)` can be declared once in the loop and reused instead of already accessing that attribute from `_currentAsset` and casting it to `IERC20` . This is just an example across the whole codebase.
6) [ ] ISectorAccountingEngine.sol#L192: `withdrawable()` returns the amount of base assets in REDEEM and in DEPOSIT sectors, not exclusively REDEEM.
7) [ ] ISubQueryEngine.sol#L56-L69: The description is incorrect. The implementation computes IDs, checks duplicate query IDs, stores the sub-query index, and emits Created. It does not move assets, account ephemeral values, or sync activation.
8) [ ] IVehicleRegistry.sol#L90-L94: Comments indicate that unauthorized vehicles return default config with unlimited cap. The actual mapping default is `VehicleMode.Automatic` with `cap.value = 0` and `cap.`

`threshold = 0`, which is noted correctly in ISectorAccountingEngine and VehicleRegistry. Unlimited cap is `type(uint256).max` and not `0`.

9) MultiVehicle.sol#L68-L69: Comments do not mention the `retrievable()` assets in `totalAssets`.

10) [ ] MultiVehicle.sol#L245-L249: Comments are not matching the actual code. The `redeemQueue.retrievable()` assets are not mentioned.

11) [ ] MultiVehicleFactory.sol#L215: the `initialCounter` input parameter in the `MultiVehicleFactory` constructor (and the respective natspec documentation) should be renamed `startingCounter` to be consistent with the other factories.

12) [ ] QueueStrategyEngine.sol#L51: The comment about holdings calculated is not correct. It describes the calculation as *"ALLOCATION shares - shares staged for redemption + expected shares from PROCESSING deposits"* which is not what the code does.

13) [ ] QueueStrategyEngine.sol#L173: Missing natSpec documentation for return values.

14) [ ] QueueStrategyEngine.sol#L275: the `_accountingEngine.asset()` asset can be stored in a local variable outside the `allocate` loop instead of always being fetched with an external call.

15) [ ] QueueStrategyEngine.sol#L311: The `vehicleHoldings` function retrieves 5 values but only one of them is used. Consider having a dedicated view function for gas optimization.

16) [ ] QueryRedeemQueue.sol#L207: There is no validation in the code that asset decimals are lower than or equal to 18.

17) [ ] QueryRedeemQueue.sol#L612-L614: the `_demandIndex` validation in the `QueryRedeemQueue._redeemable` function can be removed. The `demandId` is already validated by `_isValidDemandId` called in both the `redeem` and `redeemable` functions.

18) [ ] Sector.sol#L83-L88: rename the `queryIndex` parameter of the `Sector.toSector` function to `subQueryIndex`. That parameter refers to the subquery ID and not the query one. Perform such change also in the relative natspec.

19) [ ] SectorAccountingEngine.sol#L385: consider emitting a specific event when the `vehicleRegistry` is configured during the `__SectorAccountingEngine_init` execution to be consistent with the event emissions for the other state variable of that contract.

20) [ ] SectorAccountingEngine.sol#L452: Please add a code comment that `_unallocation.assets` are sub-vehicle shares.

21) [ ] SectorAccountingEngine.sol#L510: In `moveAssets` the `_isAuthorizedVehicle(_vehicle)` check can be removed. `_checkValidAssetMoveSector(to, false);` is already validating that if the `to` sector is a "vehicle sector", such vehicle must be authorized.

22) [ ] SectorAccountingEngine.sol#L759: `_depositSectorAssetBalance` should be renamed to `_vehicleSectorAssetBalance` or `_stagedDepositAmount` to clarify it represents assets staged in a specific vehicle's sector, not the global DEPOSIT sector balance.

23) [ ] SectorAccountingEngine.sol#L766-L782: the `DepositLimitedByCap` event should also be emitted inside the conditional branch `if (_atCap) {`.

24) [ ] SectorAccountingEngine.sol#L808-L811: Rename `_depositSectorSharesBalance` into `_vehicleSectorSharesBalance`.

25) [ ] SectorAccountingEngine.sol#L1051: rename `_isActiveVehicle` to `_isVehicleActive` to follow the external function that is calling it (`isVehicleActive`).

26) [ ] SubQueryEngine.sol: rename all the instances of `subquery` to `subQuery` to be consistent across the whole `SubQueryEngine` contract's code.

27) [ ] SubQueryEngine.sol#L71-L74: Comment indicates that *"Partial unlock: Proportional decrement"*. This is not correct, the decrement is amount based and not proportional.

28) [ ] SubQueryEngine.sol#L133 + SubQueryEngine.sol#L139: Consider tracking the `SubQuery` in both the `AccountQuery` and `UnaccountQuery` events.

29) [ ] SubQueryEngine.sol#L139: consider tracking the final `state` of the query (SETTLED/REJECTED) in the `UnaccountQuery`.

30) [ ] SubQueryEngine.sol#L145: consider renaming the `Created` event to `CreatedQuery` to be consistent with the `FinalizedQuery` event name.

31) [ ] SubQueryEngine.sol#L151: `subQueryId` of the `FinalizedQuery` should be declared as `indexed` like in the `Created` event.

32) [ ] SubQueryEngine.sol#L247-L250: consider tracking the `SubQueryEngine.withdraw` execution with an event. Note that the `msg.sender` is the `SectorAccountingEngine` but receiver is the `MultiVehicle` itself.

33) [ ] SubQueryEngine.sol#L270: rename `_queryIndex` to `_subQueryIndex` in the `SubQueryEngine.progressQuery` function.

34) [ ] SubQueryEngine.sol#L494-L501: In the `_progressRecoveringQuery` rename the `queryIndex` input parameter (and the relative natspec reference) to `subQueryIndex` and `_querySector` to `_subQuerySector`.

35) [ ] SubQueryEngine.sol#L524-L526: consider replacing the whole `.pop` loop in `SubQueryEngine._progressRecoveringQuery` with just `delete $.queryRecoveringAssets[_queryId]`.

36) [ ] SubQueryEngine.sol#L531-L534: rewrite the whole `SubQueryEngine._accountQuery` natspec. This function is **always** executed no matter what the new state is. The ephemeral accounting is **not** increased after the `resume()` operation. In general the natspec is confusing and inaccurate.

37) [ ] SubQueryEngine.sol#L535: Consider renaming the function name to a name that puts emphasis on the fact that the function maintains ephemeral accounting.

38) [ ] SubQueryEngine.sol#L620: `_unaccountQuery` is always called with `accountedAssetStore` set to `$.queryProcessingAssets`. Consider renaming the parameter to `queryProcessingAssets`.

39) [ ] SubQueryEngine.sol#L626: In the `SubQueryEngine._unaccountQuery` function remove the `if ($accountedAssets.length == 0) return;` code. It should be impossible to reach it from a valid state given that `queryProcessingAssets` is **always** modified/updated during the execution of `_progressEmptyQuery`.

40) [ ] Target.sol: consider defining an `UNLIMITED` constant variable and using it across the `Target` contract where the `type(uint256).max` value is used.

41) [ ] Target.sol#L30-L38: The `Target` library defines both an `UNLIMITED` and `unlimited` function that returns two different types of values (with two different meanings). This behavior can be confusing. Consider maybe renaming those functions like this: `UNLIMITED` to `unlimitedValue` and `unlimited` to `unlimitedTarget`.

42) [ ] VehicleRegistry.sol: The current implementation of the `VehicleRegistry` has a "general purpose name" but only supports single-asset vehicles. Consider renaming the contract to `SingleAssetVehicleRegistry` or accept an arbitrary `VehicleLib.VehicleCategory` category to be used during the `_authorize` logic when `VehicleLib.isCategory` is executed.

43) [ ] VehicleRegistry.sol#L226: The comments say that `configure()` always `VehicleConfigured`, even if the config is unchanged. However, the implementation reverts with `StateUnchanged()` if mode, cap value, and cap threshold are all unchanged.

Recommendation: Kiln should consider implementing the suggestions listed above.

Kiln: Fixed in commit d4a6bf3f (audit cleanups), PR #445 (VehicleRegistry → VehicleManager rename), PR #448 (retrievable removal).

Fixed by applying the recommendations: renames (`minSharesForAutoFulfill` , `tryAutoFulfill` , `startingCounter` , `subQuery` casing), `TargetLib.make()` factory with unlimited sanity check, satellite init events (`ParentMultiVehicleInitialized` , `BaseAssetInitialized` , ...), `owner` removal from QRQ + SAE, doc corrections (`withdrawable()` , `ISubQueryEngine` , `IVehicleManager.getConfig`), and the `VehicleRegistry → VehicleManager` rename (PR #445). The `totalAssets` comment no longer needs to mention `retrievable` — it was removed from the calculation entirely by the fix for the `QueryRedeemQueue.retrievable` finding (PR #448). Minor stylistic divergence: the suggested `UNLIMITED` constant is implemented as the functions `TargetLib.unlimitedValue()` / `unlimitedTarget()` , since the value is a sentinel encoded in a struct factory rather than a standalone constant.

Spearbit: Fixed. The suggestions have been applied.

6.3.4 Natspec issues

Severity: Informational.

Description:

1) [ ] MultiVehicleJobListing.sol#L164: The natspec comment ”Calls accountingEngine.requestWithdrawable() to unallocate from sub-vehicles” is wrong. The `MultiVehicle.feedQueryRedeemQueue` function does not internally call `accountingEngine.requestWithdrawable()` . Update the natspec with the current flow followed by the MultiVehicle.
2) [ ] ISectorAccountingEngine.sol#L144 + ISectorAccountingEngine.sol#L153: the `dispatch` function does **not** revert if the vehicle is not authorized. The internal `_dispatchRedeem` does not perform such validation and the `_dispatchDeposit` only ”early return” with state `EMPTY` . Update the natspec.
3) [ ] SubQueryEngine.sol#L68-L69: Comment is not accurate. Ephemeral accounting is not increased on `resume()` .
4) [ ] SubQueryEngine.sol#L94: `SubQueryEngineStorage.subQueryIndex` is not used to generate unique sub-query IDs. Update the natspec of the `SubQueryEngineStorage` struct.
5) [ ] SubQueryEngine.sol#L96: The `queryProcessingAssets` mapping is **always** updated with the estimation of the query operation no matter what the query state. Update the natspec of the `SubQueryEngineStorage` struct.
6) [ ] SubQueryEngine.sol#L129: The `AccountQuery` event is always emitted when the `_progressEmptyQuery` function is executed, no matter what the query state ends up being. Update the natspec.
7) [ ] SubQueryEngine.sol#L457 + SubQueryEngine.sol#L484: The `SubQueryEngine._progressUnlockingQuery` and `SubQueryEngine._progressRecoveringQuery` functions **do not** perform any loop. Update both the wrong natspec documentations.
8) [ ] IResumeFacet.sol#L34: The new STEAM state after resume **must** return `PROCESSING` according to specifications.
9) [ ] SectorAccountingEngine.sol#L481: The comment indicates that the *”from sector can be any sector.”*. This is not correct as ALLOCATION, Query and Static sectors are not allowed. It can be DEPOSIT, REDEEM, or vehicle sectors.

10) [ ] SectorAccountingEngine.sol#L517-L518: the selected natspec for the `moveShares` function is outdated and incorrect. The functions only allow movement of shares between the `ALLOCATION` and `vehicle`'s sector.

Recommendation: Review and correct the cited natspec comments and documentation strings to accurately reflect the implemented behavior. Remove unused error declarations.

Kiln: Fixed in commit 7ccdfb4d (natspec/doc accuracy bundle (4)).

Fixed by applying the recommendations — natspec corrected on `MultiVehicleJobListing.feedQueryRedeemQueue`, `ISectorAccountingEngine.dispatch` / `move`, `SubQueryEngine` (ephemeral timing, `subQueryIndex`, `queryProcessingAssets`, `AccountQuery`, `progressUnlockingQuery` / `progressRecoveringQuery`), and `IResumeFacet.resume`.

Spearbit: Verified the fixes. The natspec has been corrected as recommended.

6.3.5 Cross-contract sanity checks during MultiVehicle initialization

Severity: Informational.

Description: The `MultiVehicle` initialization is the last step of the deployment phase performed by the `MultiVehicleFactory` and could be ideal to implement additional sanity checks that validate that all the other contracts configured within the MultiVehicle and all its dependencies are indeed ”compatible” with each other.

Note that some of the getters used in the suggested sanity checks may not exist yet. In that case Kiln would be required to implement them in the relative contract.

`QueryRedeemQueue`:

- `params.redeemQueue.owner() == address(this)` → `QueryRedeemQueue` owner is the `MultiVehicle`.
- `params.redeemQueue.assetIn() == address(this)` → `QueryRedeemQueue` asset IN are the `MultiVehicle` shares.
- `params.redeemQueue.assetOut() == params.accountingEngine.asset()` → `QueryRedeemQueue` asset OUT is the `MultiVehicle`'s asset (which is the asset used by the `SectorAccountingEngine`).

`QueueStrategyEngine`:

- `params.accountingEngine.strategyEngine().multiVehicle == address(this)` → `QueueStrategyEngine` is configured with the correct `MultiVehicle`.
- `params.accountingEngine.strategyEngine().accountingEngine == params.accountingEngine` → `QueueStrategyEngine` is configured with the correct `SectorAccountingEngine`.

`SubQueryEngine`:

- `params.accountingEngine.subQueryEngine().asset() == params.accountingEngine.asset()` → `SubQueryEngine` has been configured with the correct asset.
- `params.accountingEngine.subQueryEngine().multiVehicle() == address(this)` → `SubQueryEngine` has been configured with this `MultiVehicle`.

- `params.accountingEngine.subQueryEngine().accountingEngine() == params.accountingEngine` → `SubQueryEngine` has been configured with the correct Sector Accounting Engine.

`VehicleRegistry`:

- `params.accountingEngine.vehicleRegistry().multiVehicle() == address(this)` → `VehicleRegistry` has been configured with this `MultiVehicle`.
- `params.accountingEngine.vehicleRegistry().multiVehicle() == params.accountingEngine.asset()` → `VehicleRegistry` has been configured with the correct asset.

`SectorAccountingEngine`:

- `params.accountingEngine.owner() == address(this)` → `SectorAccountingEngine` owner is the `MultiVehicle` itself.
- `params.accountingEngine.multiVehicle() == address(this)` → `SectorAccountingEngine` has been configured with this `MultiVehicle`.

Recommendation: Kiln should consider implementing the above cross-contract sanity checks during the `MultiVehicle`'s initialization (`MultiVehicleFacets.__MultiVehicle_init`) to ensure that all the direct and indirect dependencies of the `MultiVehicle` have been initialized and configured properly.

Note that this validations rely on the assumption that the MultiVehicle is the last contract initialized during the `MultiVehicleFactory` deployment phase.

Kiln: Fixed in commit 03aca2b3 (cross-contract init sanity checks (7)).

Fixed by applying the recommendation — `_checkSatelliteWiring` validates all satellite/parent/asset/manager references at MultiVehicle init, reverting `MisconfiguredMultiVehicle(satellite, reason)` on any mismatch.

Spearbit: Verified the fixes, also the owner is removed from `SectorAccountingEngineStorage` and the respecting recommendation above is not relevant.

6.3.6 `VehicleRegistry` should be refactored

Severity: Informational.

Description: The `VehicleRegistry` will benefit from a refactoring in both the security (sanity checks, redundant operations, spammy events) and code point of views.

1) The `unauthorize` function must revert if the `vehicle` has **not** been authorized.
2) The `_authorize` function must revert if the `vehicle` has already been authorized ($.vehicles[vehicle].index != 0). The `if ($.vehicles[vehicle].index == 0)` check can be removed given that this function can be executed only for not-yet authorized (or unauthorized) vehicles.
3) The `_authorize` function must revert if `vehicle` is the `_getStorage().multiVehicle` itself.
4) The `_authorize` function must emit the `VehicleConfigured` event when the vehicle's config has been initialized or changed.
5) The `_authorize` function should take the `VehicleConfig` as an input parameter and use it to initialize the vehicle's config `$.vehicles[vehicle].config`.

After the above changes we can also refactor the `authorize` and `authorizeAndConfigure` functions:

```
function authorize(IVehicle vehicle)
    external
    nonReentrant
    onlyGatedRole(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION)
{
    _authorize(vehicle, VehicleConfig({mode: VehicleMode.Automatic, cap:
    ↪  TargetLib.unlimited()}));
}

function authorizeAndConfigure(IVehicle vehicle, VehicleConfig calldata config)
    external
    nonReentrant
    onlyGatedRole(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION)
{
    _authorize(vehicle, config);
}
```

Recommendation: Kiln should implement the refactoring suggested above.

Kiln: Fixed in commit d4ec65d0 (VehicleManager auth lifecycle).

Fixed by applying the recommendation — `_authorize` takes `VehicleConfig`, reverts `CannotAuthorizeMultiVehicle` / `VehicleAlreadyAuthorized`; `unauthorize` reverts `VehicleNotAuthorized` on a missing entry; `authorize` / `authorizeAndConfigure` funnel through `_authorize`.

Spearbit: The fix has been applied. The review of the refactor (including the rename of `VehicleRegistry` to `VehicleManager`) was not completed during the engagement; it touches the authorization and configuration surface of the MultiVehicle and is listed in the appendix (see the "Focus point document") as requiring further verification.

6.3.7 Considerations and side effects relative to the assumptions for the `ready()` vehicle's behavior

Severity: Informational.

Description: All the existing "simple" `BaseVehicle`'s like `AaveV3Vehicle`, `MorphoBlueVehicle` and so on behave all the same: once the factory has deployed the vehicle and made the first deposit, the `enable()` function is executed and the vehicle is "marked" as ready.

All the existing "simple" vehicles that have been reviewed so far behave the same and do not override the `BaseVehicle.ready()` function that simply returns `BaseVehicleStore.getStorage().enabled`.

The MultiVehicle and all the contracts on which it relies (like `VehicleRegistry`, `SectorAccountingEngine`, `QueueStrategyEngine` and so on) on the **strong assumption** that once the vehicle has been "marked" as ready, it **cannot** go back to a non-ready state anymore.

While this is true for all these "simple" vehicles, the Kiln team has confirmed that **this assumption is wrong** and cannot be used as a real assumption. In the future they could introduce vehicles that have a "dynamic readiness" or change how the existing "simple" vehicles behave.

This fact has multiple side effects on the existing code that need to be taken into consideration in multiple contracts:

`VehicleRegistry`: A vehicle can be marked as "authorized" only if it's ready.

Given the mutability of the ready-state of a vehicle, the `_isAuthorized` function logic **must** be changed. The fact that the vehicle has been marked as ready in the past does not mean that it can be seen as authorized forever.

```
  function _isAuthorized(IVehicle vehicle) private view returns (bool) {
-     return _getStorage().vehicles[vehicle].index != 0;
+      return _getStorage().vehicles[vehicle].index != 0 && vehicle.ready();
  }
```

`QueueStrategyEngine`: `QueueStrategyEngine.allocate`

The `QueueStrategyEngine.allocate` function does not skip the allocation loop logic for vehicles that are not ready anymore. No changes to the code are needed if the suggestion made in the `VehicleRegistry` section is implemented given that the code is anyway checking `_accountingEngine.isAuthorizedVehicle(_currentQueueEntry.vehicle)`.

If a non-ready vehicle is added to the list of valid allocations, the `SectorAccountingEngine.deposit` functions (which call the `QueueStrategyEngine.allocate` function) will revert, reverting the whole user deposit operation.

IMPACT: LOW (the allocator manager can remove the vehicle from the `depositQueue`).

`QueueStrategyEngine.unallocate`

The `QueueStrategyEngine.allocate` function does not skip the allocation loop logic for vehicles that are not ready anymore.

If a non-ready vehicle is added to the list of valid allocations, the `SectorAccountingEngine.requestWithdrawable` function (which calls the `QueueStrategyEngine.unallocate` function) will revert, reverting the whole user redeem operation.

IMPACT: LOW (the allocator manager can remove the vehicle from the `redeemQueue`).

Kiln: Fixed in commit 6fea57ab (QSE.setQueues sanity + STEAM.md §ready() monotonicity).

The audit's premise — that `ready()` may flip back to `false` after returning `true` — is no longer permitted by the spec. STEAM.md §14 `ready()` now mandates monotonicity (`MUST NOT return false thereafter`), so the runtime-drift mitigations the finding suggested aren't needed. The `setQueues` check added in commit 6fea57ab rejects not-yet-initialized vehicles at install time, which is the only remaining failure mode.

Spearbit: Verified the fixes.

6.3.8 Document unclear or unspecified implicit assumptions

Severity: Informational.

Description:

- ☐ All the vehicles that the `MultiVehicle` is interacting with are **trusted** vehicles that behave following the STEAM standard and the ”declared” category type (single asset vehicle, etc).

Recommendation: Kiln should explicitly document the above trust assumptions.

Kiln: Fixed in commit 7a674775 (estimate() docs (54)); plus pre-existing wording in `src/docs/how-to/operate-multivehicle.md`.

Sub-vehicle trust requirements are documented: STEAM compliance and `SingleAsset` category in the authorisation section of `operate-multivehicle.md`, and recursive trust on sub-vehicle `estimate()` in `src/docs/explanation/estimate-as-accounting-input.md`.

Spearbit: Verified the fixes. The trust assumptions on sub-vehicles are now documented.

6.3.9 Consider refactoring part of the `QueueStrategyEngine.allocate` code

Severity: Informational.

Description: The whole code block inside the `if (!_effectiveTarget.isUnlimited()) {` conditional branch in the `QueueStrategyEngine.allocate` function is **the same** code/logic that is implemented by the `SectorAccountingEngine._getActiveHoldings` function.

Recommendation: Kiln should:

1) Make the `SectorAccountingEngine` expose `_getActiveHoldings` via an external function.
2) Replace the whole code with the new exposed function in `SectorAccountingEngine`.

```
if (!_effectiveTarget.isUnlimited()) {
    _vehicleSharesHoldings = _accountingEngine.getActiveHoldings(_currentQueueEntry.vehicle);

    if (_effectiveTarget.isAtTarget(_vehicleSharesHoldings)) {
        continue;
    }
}
```

Kiln: Fixed in commit eb79825e.

Fixed by applying the recommendation — `SectorAccountingEngine.vehicleEstimatedShares` exposed (alongside `vehicleSettledShares` for the settled-only consumer); `QueueStrategyEngine.allocate` consumes it instead of re-aggregating the same logic inline.

Spearbit: Verified the fixes.

6.3.10 `_createDeposit` can not compute non-zero extra shares and non-zero missing shares

Severity: Informational.

Context: MultiVehicleFacets.sol#L277-L286.

Description: The `_extraShares` and `_missingShares` computed in the `_createDeposit` can not both be non-zero.

```
uint256 _missingShares = _sharesToMint - Math.min(_sharesToMint, _totalSharesFulfilled);
uint256 _extraShares = _totalSharesFulfilled - Math.min(_sharesToMint, _totalSharesFulfilled);

// [4] Handle share balance: burn excess or mint missing shares
if (_extraShares > 0) {
    _burn(address(this), _extraShares);
}
if (_missingShares > 0) {
    _mint(address(this), _missingShares);
}
```

Recommendation: Use `if/else if` pattern instead.

```
// [4] Handle share balance: burn excess or mint missing shares
if (_totalSharesFulfilled > _sharesToMint) {
    _burn(address(this), _totalSharesFulfilled - _sharesToMint);
} else if (_sharesToMint > _totalSharesFulfilled) {
    _mint(address(this), _sharesToMint - _totalSharesFulfilled);
} // else, perfect equality so nothing to do (no shares to mint or burn)
```

Kiln: Fixed in commit 77d03ca7.

Fixed by refactor. `_createDeposit` rewritten with delta-based mint as part of a broader fix. The dual `_extra Shares` / `_missingShares` shape the finding flags no longer exists: auto-fulfill burns only the residual against `withdrawable`, and delta-based mint covers the rest with no parallel branches.

Spearbit: Fixed. The `if/else if` pattern is not required anymore.

6.3.11 `QueryRedeemQueue.redeem` could return the pending amount

Severity: Informational.

Context: MultiVehicleFacets.sol#L173-L177.

Description: The `QueryRedeemQueue.redeem()` function is used during the MultiVehicle `unlock()` logic. It returns the amount of base assets redeemed.

```
/// @return The amount of base assets (assetOut) redeemed and transferred to the caller.
function redeem(uint256 demandId) external returns (uint256);
```

As this function is only called in `MultiVehicle.unlock()` and that this unlock logic then queries the `pending()` value for the same demand, the `redeem()` function could return this pending value. This would avoid having two different external calls.

```
function unlock(Query calldata query, Id qid, uint256, uint256)
    external
    override
    returns (State, State[] memory, Asset[] memory)
{
    // ...

    // [4] Check if query has queued demand and if it's redeemable
    uint256 _demandId = $.values[qid].demandId;

    // [5] Redeem from queue if fulfillments available
    if (_demandId > 0 && $.redeemQueue.redeemable(_demandId)) {
        _totalAssetsToUnlock += $.redeemQueue.redeem(_demandId); // @audit get the amount to
        ↪  unlock
        // Check if demand fully fulfilled (pending == 0)
        if ($.redeemQueue.pending(_demandId) == 0) { // @audit queries the pending value
            $.values[qid].demandId = (_demandId = 0);
        }
    }
```

Recommendation: Modify the `QueryRedeemQueue.redeem()` function such that it also returns the remaining amount for the given demand.

Kiln: Fixed in commit a4ef2c7c.

Fixed by applying the recommendation. `QueryRedeemQueue.redeem` returns `(redeemedAssets, remainingShares)`; `MultiVehicle.unlock` consumes the tuple and drops the redundant `pending()` call.

Spearbit: Fixed.

6.3.12 Consider tracking with an event the amount not allocated/unallocated during the `deposit` and `requestWithdrawable` execution

Severity: Informational.

Description:

Deposit: The `SectorAccountingEngine.deposit` function should track with an event how much of the `amount` will not be allocated by the `$.strategyEngine.allocate` and will remain idling in the `SectorAccountingEngine` contract without producing any yield. The event could also track how much is already idling in the `DEPOSIT` sector and could be allocated via manual dispatch operations.

Redeem: The `requestWithdrawable` function is called by the `MultiVehicleFacets` when there's an amount of assets that needs to be pulled to fulfill the user's redeem request or the existing `QueryRedeemQueue` existing demand not yet fulfilled.

In this case the `amount` input represents the total amount that needs to be withdrawn. If `amount` is fulfilled by the existing balance of assets or by the expected one (in the ephemeral accounting) the function early returns. Otherwise the function tries to unallocate what's left to be fulfilled.

The current logic **does not** track via an event the part of `amount` can't be unallocated by the redeem queue of the `QueryStrategyEngine`. That event would be very useful for the Allocators that **must** fill that remaining request manually. The event should track how much of `amount - (__withdrawable + _ephemeralRedeemSectorAmount)` cannot be unalloated by the `QueryStrategyEngine.unallocate` and that needs to be manually unallocated by the Funds Managers.

Recommendation: The funds manager should monitor these events and trigger manual operations to allocate those funds to start producing yield or fill existing withdraw demands from users that have expressed their will to exit the vehicle.

Kiln: Fixed in commit 53539e88.

Fixed by applying the recommendation — `DepositPartiallyAllocated(totalAmount, allocatedAmount, residualIdle)` event added to `SectorAccountingEngine.deposit` (paired with the `RequestWithdrawableShortfall` event added for the silent liveness failure finding on the redeem side). NatSpec documents the operator recovery action.

Spearbit: Verified the fixes. The `DepositPartiallyAllocated` event now tracks the amount that could not be allocated.

6.3.13 Consider refactoring the `SectorAccountingEngine.syncVehicleActivationStatus` function for a better and more secure DX

Severity: Informational.

Description: The `SectorAccountingEngine.syncVehicleActivationStatus` function can **only** be called by the `SubqueryQueryEngine`. Because of that we suggest the following changes:

1) Remove the `active` flag, this flag should be driven by the actual "active" state of the vehicle and not passed as an input. This removed any possible error like activating a vehicle that does not have funds (and the reverse).
2) Remove the `isVehicleActive` check done by the `SubQueryEngine` before calling this function.
3) Change the logic of `syncVehicleActivationStatus` to this.

```
-  function syncVehicleActivationStatus(IBaseVehicle vehicle, bool active) external
↪   onlySubQueryEngine {
+  function syncVehicleActivationStatus(IBaseVehicle vehicle) external onlySubQueryEngine {
+     bool active = _isActiveVehicle(vehicle);
      _syncVehicleActivationStatus(vehicle, active);
  }
```

4) Now the `SubQueryEngine` does not need to check it and can just call `$.accountingEngine.syncVehicleActivationStatus(_vehicle);`

Recommendation: By implementing the suggested changes Kiln will gain the following benefits.

1) Ensure that it's impossible to activate a non-active vehicle.
2) Ensure that it's impossible to deactivate an active vehicle.
3) Avoid making an additional external call from the `SubqueryQueryEngine` to fetch the is-active boolean flag.

Kiln: Fixed in commit b9e1acb1.

Fixed by applying the recommendation — external `syncVehicleActivationStatus(vehicle)` drops the `active` arg and derives state internally via `_isVehicleActive`; SubQueryEngine call sites no longer pre-query.

Spearbit: Verified the fixes. `syncVehicleActivationStatus` now derives the activation state internally.

6.3.14 `_totalUnaccountQuery` loop refactor

Severity: Informational.

Context: SubQueryEngine.sol#L597-L598.

Description: The `_totalUnaccountQuery` function logic uses a loop with a reverse index to unaccount all ephemeral accounting. This reverse index calculation is confusing and can be refactored.

```
function _totalUnaccountQuery(IBaseVehicle vehicle, SubQuery memory subQuery, Asset[] storage
↪   $assets)
    internal
    returns (Asset[] memory)
{
    SubQueryEngineStorage storage $ = _getStorage();

    uint256 _assetsLength = $assets.length;
    for (uint256 _idx = 0; _idx < _assetsLength; ++_idx) {
        uint256 _reversedIdx = _assetsLength - 1 - _idx;
        Asset memory _currentAsset = $assets[_reversedIdx];

        $.vehicleEphemeralAccounting[vehicle][IERC20(_currentAsset.asset)] -=
        ↪  _currentAsset.value;
```

```
        $.sectorEphemeralAccounting[subQuery.settledDestination][IERC20(_currentAsset.asset)]
        ↪  -= _currentAsset.value;

        $assets[_reversedIdx].value = 0;
        $assets[_reversedIdx].asset = address(0);
        $assets.pop();
    }

    return new Asset[](0);
}
```

Recommendation: Consider making this loop calculations more readable.

```
  diff --git a/src/vehicles/multi/SubQueryEngine.sol b/src/vehicles/multi/SubQueryEngine.sol
  index 58eb0861..a91dac0e 100644
-  -- a/src/vehicles/multi/SubQueryEngine.sol
+  ++ b/src/vehicles/multi/SubQueryEngine.sol
  @@ -593,16 +593,14 @@ contract SubQueryEngine is ISubQueryEngine, ReentrancyGuardUpgradeable,
   ↪  Multical
       {
           SubQueryEngineStorage storage $ = _getStorage();

-          uint256 _assetsLength = $assets.length;
-          for (uint256 _idx = 0; _idx < _assetsLength; ++_idx) {
-              uint256 _reversedIdx = _assetsLength - 1 - _idx;
-              Asset memory _currentAsset = $assets[_reversedIdx];
+          uint256 _assetsLastIndex = $assets.length;
+          while (_assetsLastIndex > 0) {
+              _assetsLastIndex--;
+              Asset memory _currentAsset = $assets[_assetsLastIndex];

              $.vehicleEphemeralAccounting[vehicle][IERC20(_currentAsset.asset)] -=
              ↪  _currentAsset.value;
              $.sectorEphemeralAccounting[subQuery.settledDestination][IERC20(_currentAsset.a
              ↪  sset)] -= _currentAsset.value;

-              $assets[_reversedIdx].value = 0;
-              $assets[_reversedIdx].asset = address(0);
              $assets.pop();
          }
```

Kiln: Fixed in commit 5edc2aca.

Fixed by applying the recommendation. Reverse-walk rewritten as `while (remaining > 0) { --remaining; }`, dead zero-overwrites before `pop()` removed.

Spearbit: Fixed.

6.3.15 `VehicleRegistry` threshold configuration sanity checks

Severity: Informational.

Context: VehicleRegistry.sol#L246.

Description: `VehicleRegistry` accepts a `config.cap.threshold > config.cap.value` configuration.

This would lead the `effectiveValue` to always return 0 while the value is not zero.

```
// src/vehicles/multi/VehicleRegistry.sol?lines=229,249
    function configure(IVehicle vehicle, VehicleConfig calldata config)
        external
        nonReentrant
        onlyGatedRole(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION)
    {
        if (!_isAuthorized(vehicle)) {
            revert VehicleNotAuthorized(vehicle);
        }

        VehicleRegistryStorage storage $ = _getStorage();
        VehicleConfig memory _oldConfig = $.vehicles[vehicle].config;
        if (
            _oldConfig.mode == config.mode && _oldConfig.cap.value == config.cap.value
                && _oldConfig.cap.threshold == config.cap.threshold
        ) {
            revert StateUnchanged();
        }
        $.vehicles[vehicle].config = config;

        emit VehicleConfigured(vehicle, config);
    }

// src/vehicles/multi/libs/Target.sol?lines=52,57
    function effectiveValue(Target memory t) internal pure returns (uint256) {
        if (t.value == type(uint256).max) {
            return type(uint256).max;
        }
        return t.value >= t.threshold ? t.value - t.threshold : 0;
    }
```

Recommendation: Consider ensuring that `config.cap.threshold < config.cap.value`.

Kiln: Fixed in commit 5490eede.

Fixed by applying the recommendation. `TargetLib.checkTarget` invariant enforced at `TargetLib.make`, `VehicleManager.configure` / `_authorize`, and `QSE._checkQueue`; reverts `InvalidTarget(value, threshold)` when `threshold > value`. `threshold == value` is accepted (soft-pause semantic).

Spearbit: Fixed. The invariant is now checked.

6.3.16 Consider refactoring the `SectorAccountingEngine._isActiveVehicle` function

Severity: Informational.

Description: The current implementation of the `SectorAccountingEngine._isActiveVehicle` function is basically reimplementing what the `_vehicleHoldings` is doing and returning.

Consider replacing it with the values returned by `_vehicleHoldings` to reduce the code and logic to manage and avoid future possible errors where the logic diverges accidentally.

Recommendation: Kiln should consider refactoring the `_isActiveVehicle` as suggested below.

```
  function _isActiveVehicle(IBaseVehicle vehicle) internal view returns (bool) {
-       SectorAccountingEngineStorage storage $ = _getStorage();
-       ISubQueryEngine _subQueryEngine = $.subQueryEngine;
```

```
-      IERC20 _assetAddress = $.asset;
-
-      return (_getSectorAssetBalance(SectorLib.ALLOCATION, vehicle) > 0
-              || _getSectorAssetBalance(vehicle.toSector(), vehicle) > 0
-              || _getSectorAssetBalance(vehicle.toSector(), _assetAddress) > 0
-              || _subQueryEngine.vehicleEphemeralAccounting(vehicle, vehicle) > 0
-              || _subQueryEngine.vehicleEphemeralAccounting(vehicle, _assetAddress) > 0);
+     (
+         uint256 _sharesAfterUnlocks,
+         uint256 _sharesBeforeCreates,
+         uint256 _expectedSharesAfterUnlocks,
+         uint256 _expectedAssetsAfterUnlocks,
+         uint256 _assetsBeforeCreates
+     ) = _vehicleHoldings(vehicle);
+
+     return
+         _sharesAfterUnlocks > 0  ||
+         _sharesBeforeCreates > 0 ||
+         _expectedSharesAfterUnlocks > 0 ||
+         _expectedAssetsAfterUnlocks ||
+         _assetsBeforeCreates > 0;
  }
```

Kiln: Fixed in commit 219ad5fa.

The literal recommendation (reuse `_vehicleHoldings`) would have preserved an ephemeral-drain false negative documented in the finding "`_isActiveVehicle` may treat a vehicle as inactive while it has on-going queries" — if a partial unlock decrements ephemeral entries to 0 while the sub-query is still in flight, `_vehicleHoldings`-based detection would wrongly mark the vehicle inactive. Instead, the ephemeral checks in `_isVehicleActive` were replaced by a dedicated `SubQueryEngine.inFlightQueries(vehicle)` counter incremented at `createSubQuery` and decremented only on SETTLED / REJECTED. Same duplication-removal outcome as recommended here, plus closes that correctness gap.

Spearbit: Verified the fixes. The ephemeral checks were replaced by a dedicated in-flight query counter, which also removes the duplication.

6.3.17 Consider returning and "bubbling up" the result of the internal `_dispatch*` operations in the `_dispatch` function

Severity: Informational.

Description: The current implementation of the `_dispatch` function is not returning the result of the internal dispatches.

```
function _dispatch(IBaseVehicle vehicle, DispatchParams memory depositParams, DispatchParams
↪  memory redeemParams)
   internal
{
   _dispatchDeposit(vehicle, SectorLib.ALLOCATION, SectorLib.DEPOSIT, depositParams);
   _dispatchRedeem(vehicle, SectorLib.REDEEM, SectorLib.ALLOCATION, redeemParams);
}
```

Both `_dispatchDeposit` and `_dispatchRedeem` functions return the `Query` created by the underlying vehicle and the state after executing `v.create(...)` (by the `SubQueryEngine.progressQuery`.

The `_dispatch` and `dispatch` functions should both bubble up those returned values and allow the root caller to decide how to interpret the result of the dispatch operations.

Note 1) It is possible that those values could be ”empty” if those functions have early returned or not executed the query based on their own internal logic. Note 2) the ”real” state of the query in the underlying vehicle could have already changed the returned state is `PROGRESSING`.

Recommendation: Kiln should update both the `_dispatch` and `dispatch` functions to bubble up the result of the execution of `_dispatchDeposit` and `_dispatchRedeem` allowing the ”root caller” to fetch the result and act accordingly.

Kiln: Fixed in commit 601e4b4f (SAE.dispatch bubbles up (Query, State)).

Fixed by applying the recommendation — `SAE.dispatch` returns `(Query memory, State)` bubbled up from `_dispatchDeposit` / `_dispatchRedeem`.

Spearbit: Verified the fixes. `dispatch` now returns the `(Query, State)` pair produced by the internal dispatch functions.

6.3.18 `_vehicleHoldings` Comments Do Not Document Recovery Scenario for Ephemeral Accounting

Severity: Informational.

Context: SectorAccountingEngine.sol#L1091.

Description: In `SectorAccountingEngine._vehicleHoldings()`, comments 3 and 4 describe `expectedSharesAfterUnlocks` and `expectedAssetsAfterUnlocks` as values representing in-flight PROCESSING queries only. They omit the recovery scenario: when a query is in RECOVERING state, its ephemeral accounting entries are **not cleared** until the final REJECTED state is reached. This means these values also include estimates from queries currently undergoing partial recovery, which a reader would not expect from the comments alone.

Recommendation: Update comments 3 and 4 to note that ephemeral accounting is maintained throughout all partial recovery rounds and is only cleared upon final REJECTED state.

Kiln: Fixed in commit a19e548d.

Fixed by applying the recommendation — comments 3 and 4, plus the natspec `@return` lines on both `vehicleHoldings` and `_vehicleHoldings`, now document that ephemeral accounting persists through partial RECOVERING rounds and is only cleared at terminal SETTLED / REJECTED.

Spearbit: Fixed by implementing the reviewer's recommendation.

6.3.19 STEAM and vehicle specifications

Severity: Informational.

Description:

- ☒ src/docs/STEAM.md#L408: *”The returned `Asset[]` array **MUST** represent the assets that have been successfully unlocked to the `query.receiver`.”* A precision should be added to ensure that the mentioned assets successfully unlocked are in fact the assets successfully unlocked **in the current call**.
- ☒ src/docs/STEAM.md#L433: *”The returned `Asset[]` array **MUST** represent the assets that have been successfully recovered by the `query.receiver`.”* A precision should be added to ensure that the mentioned assets successfully recovered are in fact the assets successfully recovered **in the current call**.

- ☒ src/vehicles/multi/VEHICLE.md#L430-L442: The "Threshold Configuration" section refers to `setAutoRedemptionThreshold(uint256)` and the role `MULTI_VEHICLE_SET_AUTO_REDEMPTION_THRESHOLD`, but the actual implementation (MultiVehicle.sol:164-173) exposes `setThresholds(MultiVehicleStructs.Thresholds)` gated by the `MULTI_VEHICLE_SET_THRESHOLDS` role.
- ☒ src/vehicles/multi/VEHICLE.md#L466-L469: The roles table lists `FEED_QUERY_REDEEM_QUEUE`, `MULTI_VEHICLE_SET_AUTO_REDEMPTION_THRESHOLD`, and `SET_VEHICLE_AUTHORIZATION`, but the implementation uses `MULTI_VEHICLE_FEED_QUERY_REDEEM_QUEUE`, `MULTI_VEHICLE_SET_THRESHOLDS`, and `MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION` respectively (see VehicleRegistry.sol:198-258).
- ☒ src/vehicles/multi/VEHICLE.md#L531-L532: The "Vehicle Authorization" edge case section references `setVehicleAuthorization()`, which does not exist. The actual implementation (VehicleRegistry.sol:198-258) exposes four separate methods: `authorize()`, `authorizeAndConfigure()`, `configure()`, and `unauthorize()`.

Recommendation: Review and correct the cited documentation issues to accurately reflect the implemented behavior.

Kiln: Fixed in commit d682f792.

Fixed by applying the recommendation. STEAM.md `unlock` / `recover` precision on "current call"; VEHICLE.md role names corrected (`MULTI_VEHICLE_*` constants); `setAutoRedemptionThreshold` → `setThresholds`; `setVehicleAuthorization` → `authorize` / `authorizeAndConfigure` / `configure` / `unauthorize`. Propagated to `src/docs/reference/roles.md`.

Spearbit: Fixed.

6.3.20 `Target.remainingCapacity` should be used for capacity calculations

Severity: Informational.

Context: QueueStrategyEngine.sol#L263, SectorAccountingEngine.sol#L1225-L1231.

Description: `_computeCapLimitedMaxDepositable()` manually checks `_cap.isAtTarget(_activeHoldings)` and then computes remaining capacity as `cap - _activeHoldings`.

However, `TargetLib` already defines `remainingCapacity()` as the canonical helper for this calculation, including the zero-capacity case when holdings are at or above `t.value`.

```
function _computeCapLimitedMaxDepositable(IBaseVehicle vehicle, uint256
↪  underlyingMaxDepositable)
    internal
    view
    returns (uint256 capLimitedMax, uint256 cap, bool atCap)
{
    // ...
    if (!_cap.isUnlimited()) {
        // ...
        uint256 _remainingCapacityShares = cap - _activeHoldings; // @audit use
        ↪  remainingCapacity() instead
```

This `Target.remainingCapacity()` function could also be used in other locations:

- `QueueStrategyEngine.allocate`
- `SectorAccountingEngine._enforceRebalanceCap`

- `SectorAccountingEngine._dispatchDeposit`

Recommendation: Use the `Target.remainingCapacity()` helper function for this calculation.

Kiln: Fixed in commit 76584db4.

Fixed by applying the recommendation — `TargetLib.remainingCapacity(target, holdings)` helper consolidated; consumed by `SAE._computeCapLimitedMaxDepositable` and `QSE.allocate`. `_enforceRebalanceCap` mentioned in the finding had been removed earlier (no longer exists).

Spearbit: Fixed. The `remainingCapacity()` function is now being used.

6.3.21 Threshold tolerance should only be used during allocation calculation

Severity: Informational.

Context: SectorAccountingEngine.sol#L1228-L1230.

Description: In `SectorAccountingEngine._computeCapLimitedMaxDepositable`, the `isAtTarget` function is used. This function checks if the active holdings are already at cap within the threshold tolerance.

Considering that the threshold tolerance is used during the allocation time, this threshold is not needed in this function used during the `_dispatchDeposit` flow.

Recommendation: Consider checking the active holdings against the `cap.value` without the threshold tolerance during the `_dispatchDeposit` flow.

Kiln: Acknowledged. This is an intentional behavior.

The threshold-band-as-at-cap semantic is applied consistently across allocation and dispatch via `Target.remainingCapacity / isAtTarget`. The threshold expresses the operator's ”don't bother topping up below this slack” policy, and respecting it on the dispatch path avoids churn (a dispatch of a few shares of headroom that the next op would no-op anyway). Operators wanting exact-cap dispatch policy set threshold = 0.

Spearbit: Acknowledged.

6.3.22 `BaseVehicle._payout()` Does Not Guard Against Zero-Amount Transfers

Severity: Informational.

Context: BaseVehicle.sol#L547.

Description: In `BaseVehicle._payout()`, the for loop iterates over all paid assets and calls `safeTransfer` for each one without checking if `_payoutAmount > 0`. While this is a rare edge case, non-standard ERC20 tokens that revert on zero-value transfers would cause the entire `unlock()` or `recover()` transaction to revert when any single asset in a multi-asset payout has a zero amount.

Recommendation: Add a `_payoutAmount > 0` guard before the transfer calls inside the loop to skip zero-value payouts and avoid compatibility issues with non-standard ERC20 tokens.

Kiln: Fixed in commit 1f06c1ba (`BaseVehicle._payout` skips zero-value transfers).

Fixed by applying the recommendation — the receiver `safeTransfer` + `Pushed` event are wrapped in `if (_payoutAmount != 0)`, mirroring the existing fee-branch guard.

Spearbit: Fixed by implementing the reviewer's recommendation.

6.3.23 `moveShares` action can have different outcomes

Severity: Informational.

Context: SectorAccountingEngine.sol#L523-L537.

Description: The `moveShares` function allows a `MULTI_VEHICLE_MOVE_SHARES` role entity to move shares. The `_checkValidSharesMoveSector` enforces this move to be from the `ALLOCATION` sector to a `Vehicle` sector or vice-versa.

In case of a `moveShares` call to move ALLOCATION ⇒ Vehicle.toSector(), the next call to `_dispatchRedeem` will define which sector the funds will be accounted in after completion of the underlying redeem query.

The possible outcomes for the next sector are:

- `REDEEM` in case of a `requestWithdrawable` call.
- `REDEEM` in case of a `_dispatch` call (called from the `deposit` and `dispatch` external functions).
- `to.toSector()` in case of a `rebalance`.

Recommendation: First, the `_checkValidSharesMoveSector` could be simplified as it has a lot of code to only accept 2 types of sector.

Second, it should be documented that the assets redeemed after an `ALLOCATION => VEHICLE` move depend on the next call being executed.

Kiln: Fixed in commit 55b75cd3.

The outcome-ambiguity recommendation is addressed by the `move()` + `dispatch()` consolidation: the destination is now an explicit `settledDestination` parameter on the follow-up `dispatch`, so what was an implicit consequence of the next call is now a caller decision. Docs in `src/docs/how-to/operate-multivehicle.md`, `src/vehicles/multi/VEHICLE.md`, and `ISectorAccountingEngine` NatSpec describe the patterns end-to-end. The helper simplification recommendation was deliberately not applied — the per-class revert taxonomy (distinct errors for `isVehicleSector` / `isStaticSector` / `isQuerySector`) is kept because it gives callers a precise error per failure class rather than a generic ”invalid sector”.

Spearbit: Fixed. The asset manager now uses the `move()` + `dispatch()` pattern. This makes the logic deterministic for the asset manager.

6.3.24 Async Redeem Funds Received After Queue Fulfillment By Deposits Sit Idle in REDEEM Sector

Severity: Informational.

Description: When an async redeem request is created, `requestWithdrawable()` initiates unallocation from sub-vehicles. If a new deposit fulfills the queue demand before those sub-vehicle redeem queries settle, the user's redemption is satisfied without consuming the in-flight redemption. When those queries eventually settle, the returned assets land in the REDEEM sector with no automatic trigger to re-allocate them — they sit idle until an admin manually calls `moveAssets(REDEEM, DEPOSIT)`.

Recommendation: Admins should track settlement events and manually move idle REDEEM sector assets back to DEPOSIT.

Kiln: Fixed in commit 77d03ca (RESERVED sector + DEPOSIT/REDEEM → AVAILABLE merger).

Fixed via refactor — DEPOSIT and REDEEM sectors merged into a single AVAILABLE. Sub-vehicle redeem proceeds default to AVAILABLE, which `withdrawable()` / auto-fulfill / queue strategy all consume directly — no separate `move` step needed for them to be re-allocatable on the next deposit. RESERVED is the dedicated operator-park sector and only receives funds via explicit `move()`.

Spearbit: Fixed by introducing a new sector - AVAILABLE that merges both REDEEM and DEPOSIT so no need to manually move between the two.

6.3.25 Keeper assumptions

Severity: Informational.

Description: The `SubQueryEngine` emits events specifically built for an off-chain keeper. Multiple assumptions are made on this keeper software.

Keeper can handle events for non-existing jobs: `JobCancelled` and `JobDone` could be emitted with job identifiers for which no `JobStarted` event was emitted.

Keeper should not spawn one job per event: Keeper should spawn a single job per job identifier that has emitted an event. Otherwise, multiple jobs for the same job identifier could be created.

Recommendation: Kiln should design the keeper software with these assumptions.

Kiln: Acknowledged.

The two assumptions (handle `JobCancelled` / `JobDone` without prior `JobStarted`; dedupe per `jobId`) constrain the off-chain keeper bot (`tools/keeper/`), not the on-chain contracts. The contract surface (`KeeperLib.startJob` / `stopJob` / `cancelJob`) emits one event per call and treats keeper events as advisory notifications backed by a pull-based `statusTarget` poll. No contract change resolves or invalidates these assumptions. The keeper-software design owns the dedupe + out-of-order handling.

Spearbit: Acknowledged.

6.3.26 `estimate()` security considerations

Severity: Informational.

Context: SubQueryEngine.sol#L438.

Description: The STEAM.md specifications mention the following about estimate():

6. `estimate(Asset[] calldata assets, Mode mode, EstimationType estimationType) external view returns (Asset[] memory)` : **Description** Provides a **non-binding estimation** of output or input assets given current on-chain conditions and fees. The returned values are approximations and **MUST NOT** be treated as guaranteed or exact amounts by integrating contracts or protocols. The estimation reflects conditions at the time of the call. However, the moment the underlying deposit or redeem is effectively performed on the target system may differ from the moment `create()` is called — which can alter exchange rates, available liquidity, and the overall outcome. This timing gap is vehicle-specific and depends entirely on the internal implementation logic of each Vehicle (e.g., async settlement, batching, cooldown periods, external keeper processing). **Parameters**.
   - `assets` : The asset array for the estimation.
   - `mode` : The mode of estimation (`DEPOSIT` or `REDEEM`).
   - `estimationType` : The type of estimation using the `EstimationType` enum:
     - `INPUT` : Estimate the input assets required to receive the provided assets.
     - `OUTPUT` : Estimate the output assets receivable for the provided assets.
   - **MUST** return an array of `Asset` structs representing the estimated assets.
   - **MUST** be a `view` function (no state modifications).
   - **MUST** ignore the current state of any specific query; the estimation **MUST** be computed based on current market conditions and Vehicle parameters.
   - **SHOULD** provide realistic estimates, though exact results are not guaranteed and actual execution may differ due to slippage, sandwich attacks, oracle updates, interest accrual, fee changes, or other state changes between the estimation and the actual transaction. ⚠ **Integrator Warning**.

The `estimate` function is analogous to ERC-4626's `previewDeposit` / `previewRedeem` — it provides a best-effort preview, not a commitment. Integrating contracts and protocols **MUST**:

1. **Implement fault tolerance** around the returned values (e.g., slippage bounds, minimum output checks).
2. **Never use `estimate` as an oracle** or source of truth for pricing decisions.
3. **Validate actual execution results** against expected minimums rather than relying on the estimation.
4. **Account for the time gap** between calling `estimate` and the actual execution on the target system — the rate at which the operation is settled may differ from the rate at the time of estimation, depending on the Vehicle's internal implementation.

However the MultiVehicle assets per share rate depends on `estimate()` because the ”ephemeral” output amount is tracked and used for calculations in the `totalAssets()` flow (through `_vehicleHoldings()`). The code implementation and the specifications do not match. Specifications warns integrators that `estimate()` is not reliable but the code uses it.

In practice, the `estimate()` function brings side-effects including imprecisions to the `totalAssets()` calculations. Any alternative approach like avoiding estimation could also brings these side-effects. Overall, one of the side-effect that can not easily be erased is the ”assets/share” rate drop/increase. This opens the door to MEV attack vectors. It is worth noting `estimate()` should be ”precise enough” to make these drops/increases low enough such that MEV vectors are too low to be profitable.

Recommendation: Kiln should review the `estimate()` specifications. Pros and cons of the chosen `estimate()` approach should be documented. Especially, the MEV attack vectors should be documented.

Kiln: Fixed in commit 7a674775.

Fixed by applying the recommendation. New `src/docs/explanation/estimate-as-accounting-input.md` covers why MultiVehicle consumes sub-vehicle `estimate()` in its accounting pipeline, the trade-offs (truthful NAV

vs recursive trust + estimate volatility), the MEV vectors this exposes, and existing mitigations. STEAM.md and VEHICLE.md cross-reference it. VEHICLE.md adds an explicit "MUST NOT be used as an external pricing oracle" warning on `totalAssets()`.

Spearbit: Fixed through documentation.

6.3.27 `QueryRedeemQueue` security considerations

Severity: Informational.

Context: QueryRedeemQueue.sol#L53-L54.

Description: The `QueryRedeemQueue` accumulates demands when low liquidity is available. These demands are meant to be fulfilled in a First-In-First-Out (FIFO) order when liquidity is available again. Liquidity can come from user deposits or from underlying vehicle redeems through automatic deallocations or asset manager operations.

Demands in the queue are exposed to an infinite risk/ratio rate.

Demands in the `QueryRedeemQueue` can still be acquiring yield from underlying vehicles. However, any yield acquired while the demand is in the queue will be redistributed to MultiVehicle share holders.

This basically means that user demands in the `QueryRedeemQueue` are not acquiring any rewards while still being exposed to DeFi risks. This leads to an infinite risk/ratio rate.

Demands fulfillment depend on the asset manager.

The asset manager can configure the MultiVehicle such that there is no available liquidity for redeems. The MultiVehicle manager can:

- Avoid deallocations by setting an empty redeem queue in `QueueStrategyEngine` or redeem queue targets that are `type(uint256).max`.
- Deny auto-redemptions during deposits by setting `type(uint256).max` as the `minSharesForAutoRedemption` threshold.

In such state, users are locked into the MultiVehicle and can't redeem. They are stuck in a situation where they are exposed to an infinite risk/ratio rate.

Recommendation: Users must be warned about the side-effects that the asset manager and the query redeem queue can have.

Kiln: Partially fixed in commit 46c869c1.

The loss-exposure half of the warning is documented by the documentation pass for the async-redeem slippage finding: VEHICLE.md "Slippage caveat" callout, the `QueryRedeemQueue` component bullet, and `QueryRedeemQueue.demand()` / `MultiVehicleFacets._createRedeem` NatSpec all state that sub-vehicle losses, accrued fees, and rate decay between create and fulfillment are socialised into the queued demand. The remaining halves — the yield-attribution asymmetry while queued, and the operator-configurable lockout (empty redeem queue / unlimited-target redeem entries / `minSharesForAutoFulfill = type(uint256).max`) — are accepted as a known trust assumption on the `MULTI_VEHICLE_SET_QUEUES` / `MULTI_VEHICLE_SET_THRESHOLDS` role-holders rather than reframed as user-facing warnings.

Spearbit: Partially fixed. Documentation about this issue has been added.

6.3.28 MultiVehicle does not support partial success from queries in underlying vehicles

Severity: Informational.

Context: SubQueryEngine.sol#L523.

Description: According to STEAM.md specifications about recover():

> Allows the receiver to recover assets or shares after a failed or partial process. ...
> - **MUST** return unallocated or partially allocated assets/shares to the `query.receiver`.

It specifies that *"partially allocated assets/shares"* are supported. The specifications do not exclude from the standard a scenario in which only X% of a query is successful and the rest is recoverable. In such scenario, both vehicle shares and assets would be recovered through the `recover()` function.

However, the MultiVehicle current code does not support such scenario. For example, a partial redeem would lead to storing assets in the `ALLOCATION` sector. These funds would not be accounted in `MultiVehicle.totalAssets()` because assets are supposed to be in the `ALLOCATION` sector (only shares). This scenario would break the `totalAssets()` value and the calculated assets per share rate.

Scenario:

- MultiVehicle creates a 100 USDC deposit request.
- Vehicle succeeds to allocate 60 USDC to underlying protocol so it mints 60 vehicle shares to MultiVehicle.
- The remaining 40 USDC are not allocated.
- `recover()` returns `[Asset{token: USDC, value: 40}, Asset{token: vehicle shares, value: 60}]`.

At the MultiVehicle level, vehicle shares are stored in the `DEPOSIT` sector (rejected deposit query returns funds to this sector). These funds would be not accounted in `totalAssets()` and be locked as `moveShares()` do not accept `DEPOSIT` sector.

Recommendation: The standard does not exclude these scenarios but the MultiVehicle does. Consider documenting the assumption that MultiVehicle will not be used with "partial success" underlying vehicles.

Kiln: Fixed in commit 421cafdc and commit 768f3597.

Fixed by a stronger mitigation than the recommendation. The partial-success scenario is now structurally impossible. Commit 421cafdc (PR 449) introduced `QueryRegistry` as the sole state writer for vehicle queries; commit 768f3597 extended it with success/failure branch isolation via a packed `Outcome { PENDING, SUCCESS, FAILURE }` field on each `QueryRecord`. The first transition into UNLOCKING locks SUCCESS, the first into RECOVERING locks FAILURE; cross-branch transitions revert `OutcomeLocked(qid, current, attempted)`. A misbehaving sub-vehicle attempting to mix shares + assets across branches is rejected at the registry layer rather than requiring MultiVehicle to handle it gracefully.

Spearbit: Partially resolved. Branch isolation is enforced, but a single `RECOVERING` outcome returning both assets and shares (permitted by the spec as "partially allocated") is still not handled: the shares land in the `DEPOSIT` sector and are not counted in `totalAssets()`. This needs a real mitigation or a documented assumption that partial-success vehicles are unsupported. Kiln submitted a documentation follow-up after the engagement (PR 446, commit 6403be2) that has not been reviewed by Spearbit; the item is listed in the appendix (see the "Focus point document").

6.3.29 `_isActiveVehicle` may treat a vehicle as inactive while it has on-going queries

Severity: Informational.

Context: SectorAccountingEngine.sol#L1059-L1060.

Description: The `_isActiveVehicle` function is used to determine if a vehicle is active or not by getting the amount of assets and shares in various sectors and ephemeral accounting.

```
function _isActiveVehicle(IBaseVehicle vehicle) internal view returns (bool) {
    SectorAccountingEngineStorage storage $ = _getStorage();
    ISubQueryEngine _subQueryEngine = $.subQueryEngine;
    IERC20 _assetAddress = $.asset;

    return (_getSectorAssetBalance(SectorLib.ALLOCATION, vehicle) > 0
            || _getSectorAssetBalance(vehicle.toSector(), vehicle) > 0
            || _getSectorAssetBalance(vehicle.toSector(), _assetAddress) > 0
            || _subQueryEngine.vehicleEphemeralAccounting(vehicle, vehicle) > 0
            || _subQueryEngine.vehicleEphemeralAccounting(vehicle, _assetAddress) > 0);
}
```

However, this function may return `false` while there are on-going `UNLOCKING` or `PROCESSING` queries in a vehicle. This is possible when the unlocked amount is greater than the ephemeral accounting, reducing ephemeral accounting to zero while there are still funds being unlocked.

Note: In the current state of the codebase, it does not have impact.

Recommendation: At least document the edge-case. This could also be fixed by checking if there are any on-going queries.

Kiln: Fixed in commit 219ad5fa.

Fixed by applying the stricter mitigation. `SubQueryEngineStorage.inFlightQueries(vehicle)` counter incremented in `createSubQuery`, decremented only on SETTLED / REJECTED. `_isVehicleActive` reads this counter instead of the ephemeral-mapping check, strictly stronger than the original and immune to drain-window false negatives.

Spearbit: Fixed. A per-vehicle in-flight count has been implemented and is checked instead of the vehicle ephemeral accounting check.

6.3.30 `SectorAccountEngine` uses zero minimum output as slippage parameter

Severity: Informational.

Context: SectorAccountingEngine.sol#L425-L426.

Description: The `SectorAccountingEngine` configures the slippage parameter to zero in automatic actions like `deposit` and `requestWithdrawable`.

While the underlying vehicles are trusted, this lack of slippage could potentially lead to loss of funds.

```
function deposit(uint256 amount, bool allocate)
    external
    onlyOwnerOrGatedRole(Roles.MULTI_VEHICLE_DEPOSIT)
    nonReentrant
{
    // ...
    if (allocate) {
        IQueueStrategyEngine.Allocation[] memory _allocations =
            $.strategyEngine.allocate(_getSectorAssetBalance(SectorLib.DEPOSIT, _asset));
        for (uint256 _idx = 0; _idx < _allocations.length; ++_idx) {
            IQueueStrategyEngine.Allocation memory _allocation = _allocations[_idx];
            _moveFromSector(SectorLib.DEPOSIT, _allocation.vehicle.toSector(),
            ↪  _allocation.assets);
            _syncVehicleActivationStatus(_allocation.vehicle, true);
```

```
            _dispatch(
                _allocation.vehicle,
                DispatchParams({minOutput: 0, data: ""}), // @audit zero slippage protection
                DispatchParams({minOutput: 0, data: ""})
            );
        }
    }
}
```

Recommendation: A ”minimum rate” parameter could be implemented such that automatic queries are not executed when slippage amount is too high (for example due to an external loss).

This would let the asset manager able to do the allocation if required, while avoiding to automatically allocate when the outcome is risky.

Kiln: Acknowledged.

Auto-dispatch (`deposit(allocate=true)` , `requestWithdrawable`) fans out queries with `minOutput = 0` to trusted sub-vehicles whose composition is itself the operator's slippage policy. The asset manager chooses which vehicles enter the deposit / redeem queues. A sub-vehicle exhibiting unexpected loss is handled at the vehicle level (remove from queue, reconfigure cap, unauthorize) rather than via a per-engine slippage knob. User-facing `MultiVehicle.create` retains explicit `output.value` slippage on both deposit and redeem paths.

Spearbit: Acknowledged.

6.3.31 Pending redeem accounting can cause withdrawal liquidity to be overestimated

Severity: Informational.

Context: SectorAccountingEngine.sol#L442-L447.

Description: `SectorAccountingEngine.requestWithdrawable()` short-circuits when the requested amount is covered by: `withdrawable + sectorEphemeralAccounting(REDEEM, asset)` . This means the system treats in-flight redeem queries targeting the `REDEEM` sector as if they will become available withdrawal liquidity.

This is optimistic for two reasons:

- The pending redeem query may not settle and may end in the rejected sector instead.
- The pending redeem may have been created for another operational reason, such as an admin/manual dispatch, not specifically to satisfy the current user withdrawal demand.

As a result, `_createRedeem()` may avoid requesting additional unallocation from underlying vehicles even though the current withdrawal request and existing query redeem queue will not actually be covered by settled assets.

Additionally, the pending redeem once settled could be used to fulfill a `moveAssets` operation executed by the asset manager.

Recommendation: Do not rely on generic `REDEEM` sector ephemeral accounting unless it is known to correspond to withdrawal liquidity.

A safer design is to separate redeem intents more explicitly by tracking redeem queries created for withdrawal fulfillment in a dedicated sector and tracking admin/manual/rebalance redeem queries separately.

Only let `requestWithdrawable()` short-circuit based on pending redeems that are specifically intended to satisfy withdrawal demand.

Kiln: Partially fixed in commit 77d03ca7.

Concern #2 (operator-routing pollution) is structurally resolved: the refactor merged the static DEPOSIT and REDEEM sectors into AVAILABLE and added RESERVED for operator park flows. The per-sector ephemeral bucket is keyed on `subQuery.settledDestination`, so only redeems contractually destined for AVAILABLE inflate the short-circuit; operator-routed redeems (rebalance to vehicle-sector, park to RESERVED) are correctly excluded. Concern #1 (in-flight redeem may end REJECTED) is accepted as a bounded residual — the optimism window spans only the PROCESSING/UNLOCKING lifetime of an async redeem, and the next `requestWithdrawable` self-corrects once the rejected ephemeral is decremented.

Spearbit: Partially fixed and acknowledged. The new `AVAILABLE` sector is used. The ephemeral accounting to this sector is still used, which is optimistic. The `RESERVED` sector is not accounted as part of the withdrawal liquidity.

6.3.32 `_dispatchDeposit` and `_dispatchRedeem` silent fail behavior are unclear and not documented

Severity: Informational.

Description: In the current implementation of the `_dispatchDeposit` and `_dispatchRedeem` there are scenarios where the dispatch operation will "silently fail" without creating and executing any vehicle's query.

In those cases the `_dispatch*` operation will early return an "empty Query" object and the dispatch state `State.EMPTY`.

The side effects and the consequences of these behaviors depend mostly on the logic, accounting and share/asset movement that has been performed by the "root caller".

`deposit` flow: This function can be called in two instances:

- A supplier has created a `DEPOSIT` query in the `MultiVehicle` and deposited `amount` of assets.
- An authed user with the `MULTI_VEHICLE_DEPOSIT` role "donates" `amount` of liquidity to the `MultiVehicle` and executes `deposit` with the `allocate` input parameter set to `true`.

When the `_dispatchDeposit`, executed by `_dispatch`, "silently fails," what has been moved from the `DEPOSIT` sector to the vehicle's sector will remain **idling** without generating any yield for the `MultiVehicle`.

To "unlock" those funds and start generating yield we have 3 options:

1) Another user deposits funds and the same vehicle is selected from the `StrategyQueueEngine`. This time the "silent fail" conditions have been resolved and the dispatch (full or partial) is executed correctly.
2) The Asset Manager manually executes the `dispatch(vehicle)` function hoping that the criteria of the "silent fail" have been resolved.
3) The Asset Manager manually moves the assets back to the `DEPOSIT` sector (to be used by other vehicles) or directly to another Vehicle (to then trigger a manual dispatch).

`requestWithdrawable` flow: This function is executed by the `MultiVehicleFacets._createRedeem` function executed when a user creates a `REDEEM` query on the `MultiVehicle` to redeem `sharesToRedeem` and withdraw the corresponding amount of assets.

`requestWithdrawable` is called with the amount of assets that the `SectorAccountingEngine` needs to pull from what is "instantly withdrawable" or "will be instantly withdrawable" (when from the ephemeral accounting). If that balance is **not enough** it will try to unallocate what's needed from the redeem queue of vehicles selected by the `QueueStrategyEngine`.

If the `_dispatchRedeem` silently fails the vehicle's shares that have been moved from the `ALLOCATION` sector will remain "idling" in the Vehicle Sector.

To ”unlock” those shares we have these options:

1) Another user triggers a redeem operation that will select from the redeem queue the **same** vehicle. This time the ”silent fail” conditions have been resolved and the dispatch (full or partial) is executed correctly.
2) The Asset Manager manually moves the shares from the Vehicle's sector back to the `ALLOCATION` sector.
3) The Asset Manager manually triggers the `dispatch(vehicle)` function hoping that the criteria of the ”silent fail” have been resolved.

Recommendation: Kiln should evaluate the current behavior and explicitly document it to allow the Asset Managers to be aware of the actions that need to be taken to unlock those funds.

Kiln should also consider if these ”silent failure” cases should be actively managed by the ”root caller” functions to revert the idle shares into the original sector (and roll back all the accounting changes made).

Kiln: Fixed in commits commit 55b75cd3 (PR 453), commit 53539e88 (residual events and operator how-to) and commit 6877449e.

Silent fails are observable and recoverable via four structured events (`DepositPartiallyAllocated` , `RequestWithdrawableShortfall` , `LimitedDeposit` , `LimitedRedeem`) and one revert error (`EmptyStrictDispatch` for the external `dispatch()` strict mode). The new ”Handle stranded sector balances” subsection in `src/docs/how-to/operate-multivehicle.md` (added in commit 53539e88) consolidates the four events plus the three recovery patterns (wait for organic activity / retry `dispatch` / roll back via `move`) plus the `RequestWithdrawableShortfall` end-to-end recovery flow. Non-strict internal callers keep soft-fail behavior by design.

Spearbit: Verified the fixes. The silent-fail behavior is now observable through dedicated events and documented in the operator how-to.

7 Appendix

7.1 Engagement structure

The review was split into three consecutive engagements since the codebase is large:

1. **Phase 1** — the base vehicle framework and the yield-source vehicles (Aave v3, Compound v3, ERC-4626, Morpho Blue), along with the factories and shared infrastructure.
2. **Phase 2** — the MultiVehicle system.
3. **Phase 3** — the Conduit layer, plus end-to-end testing spanning the components of all three phases.

The exact scope of each part is listed in the corresponding report. One thing worth noting: the end-to-end testing in phase 3 consumed a significantly larger share of the engagement than the contract review itself, as expected given the sheer number of scenarios and configurations the composed system supports.

The project's own documentation is sufficient for understanding the codebase, so this document does not re-explain the architecture; it focuses on the areas that deserve further discussion or attention, as they may lead to real issues.

> **Naming note:** during the phase 2/3 fix review, the `VehicleRegistry` component was refactored and renamed to `VehicleManager` (see the finding "`VehicleRegistry` should be refactored"). Both names refer to the same component; the reports use the name that was current at the time of writing.

7.2 Focus areas

7.2.1 Open or partially resolved findings

The findings below were addressed by Kiln during or after the engagement, but Spearbit's review of the corresponding changes could not be completed within the engagement. They are marked as *Partially resolved* or *Acknowledged* in the findings section and require further verification in the next audit.

- **Problems and side effects of the `QueryRedeemQueue.retrievable` behavior (Medium):** Determine whether removing `retrievable` from `_totalAssets()` and routing it through the role-gated `VehicleManager.retrieveQueryRedeemQueueAssets` resolves the issue or merely relocates it. Retrieval still happens inside the `startOngoingFeeHandling` / `finishOngoingFeeHandling` flow (performance fees can still be lost) and the surplus remains observable on-chain (still sandwichable). Kiln has accepted this residual impact (Acknowledged) and may revisit the surplus distribution mechanism in a future release; the next audit should confirm that the residual risk remains bounded as described in Kiln's response.
- **Queued redeem demand distorts the child MultiVehicle's NAV; a parent MultiVehicle deposits at an understated share price (Medium):** Confirm whether the child MV's reported NAV correctly nets out queued-but-unfulfilled redeem demands. While a queue claim is outstanding, the child reports `totalAssets / totalSupply` without netting the queued `maxAmountOut`, so a parent MV holding child shares undervalues its position, and new parent depositors mint at an understated share price (over-minting at existing holders' expense). Verify that the child's NAV accounts for outstanding queued claims and that the parent's `totalAssets` is not understated during the queue window.
- **`QueueStrategyEngine.allocate` can allocate beyond the cap target (Low):** Verify the per-vehicle cap is enforced *cumulatively* across duplicate `depositQueue` entries. The `isAtTarget(_vehicleSharesHoldings)` skip check does not subtract shares hypothetically committed in earlier iterations for the same vehicle, so holdings can exceed `_effectiveTarget`. Confirm the `VehicleCommitment` accumulator deducts prior in-call commitments *before* the check, not only after. Kiln submitted a follow-up fix after the engagement (PR 446, commit eb5a540) that has not been reviewed by Spearbit.
- **`QueueStrategyEngine.unallocate` can unallocate beyond the cap target's upper bound (Low):** The same pattern on the redeem side. Confirm the `_vehicleSharesHoldings <= _effectiveTarget` skip

check accounts for shares hypothetically redeemed in earlier iterations of a duplicate vehicle entry, so that `unallocate` cannot push holdings below the configured `upperBound`. Kiln submitted a follow-up fix after the engagement (PR 446, commit 8cf4bdf) that has not been reviewed by Spearbit.

- **"Manual" dispatches can ignore the `minOutput` expressed in `DispatchParams` (Low):** The core finding is addressed by the `DispatchParams.amount` cap, and Kiln agreed that previously held shares must be considered when checking the per-vehicle targets. A follow-up fix was submitted after the engagement (PR 446, commit 0e1e045) but has not been reviewed by Spearbit. Verify that follow-up, and that the slippage path cannot be satisfied by stale `_getSectorAssetBalance` contents — particularly when `type(uint256).max` is used.
- **`_dispatchDeposit()` / `_dispatchRedeem()` early returns leave VEHICLE-sector assets stranded (Low):** Partially resolved. The `LimitedDeposit` / `LimitedRedeem` events cover the at-cap and `maxDeposit` / `maxRedeem` = 0 paths. Still open: the `!_isAuthorizedVehicle()` + `strict = false` branch (no event emitted, assets stranded) and the reachability of the `_resolvedAmount = 0` path. Also assess whether the `_allocations` loop should `continue` when `_allocation.asset.value` is 0. Kiln submitted a follow-up fix after the engagement (PR 446, commit 8e3e308) that has not been reviewed by Spearbit.
- **MultiVehicle does not support partial success from queries in underlying vehicles (Informational):** The `QueryRegistry` Outcome lock (reverting on cross-branch transitions) does not cover this case. The scenario is a single `RECOVERING` outcome in which `recover()` returns *both* assets and shares — which the STEAM specification permits. Check how the MultiVehicle accounts for shares stranded in the `DEPOSIT` sector (uncounted in `totalAssets()`, not movable via `moveShares()`). Partially resolved: a real mitigation or a documented unsupported-behavior assumption is still required. Kiln submitted a documentation follow-up after the engagement (PR 446, commit 6403be2) that has not been reviewed by Spearbit.

7.2.2 Fix-review follow-ups

- **`VehicleRegistry` should be refactored:** The refactor (including the rename to `VehicleManager`) was applied during the fix review, but its review was not completed within the engagement. Verify that the refactor itself does not introduce new errors — it touches the authorization and configuration surface of the MultiVehicle.
- **MultiVehicle's `totalAssets()` ignores sub-vehicle redeem fees, inflating share price and fees:** The fix has been applied, but the review of the new fee-aware `totalAssets()` was not completed within the engagement; it warrants particular attention in the next audit to confirm it does not introduce unintended behaviour. Specifically, verify that using the post-fee redeem estimate for sub-vehicle shares in `totalAssets()` does not cause over-minting in the DEPOSIT flow: confirm that the delta-based mint correctly accounts for the now-lower, fee-adjusted NAV so that deposits do not mint an excess number of shares. One possible fix is to use different variants of `totalAssets` for the deposit and redeem paths, depending on where it is consumed.
- **Automatic allocations to vehicles with deposit fees break share fairness (follow-up):** If assets are already present in the `vehicle.toSector()` sector before a user deposits (e.g. an admin moved funds without dispatching) **and** the vehicle charges deposit fees, then `_dispatchDeposit` is called with `type(uint256).max` and allocates *all* funds available in the sector. Under the new behavior — where the user is credited the difference in total assets — the user bears the deposit fees paid on funds they did not bring. This led to railnetorg/hangar#446, which has not been reviewed by Spearbit and should be verified in the next audit.

7.2.3 General caution

The fixes introduced for phases 2 and 3 were broad, and several of them change shared accounting paths. They may have side effects in parts of the code that were reviewed before the fixes existed. The next audit should put significant focus on regression around these areas rather than treating previously reviewed code as settled.

7.3 End-to-end testing: configurations exercised

The repository's own invariant and fork suites exercise the following system topologies, which formed the baseline for the end-to-end testing effort:

- **Single vehicles** — all vehicle types under one harness.
- **MultiVehicle with two sub-vehicles**.
- **MultiVehicle with mixed synchronous/asynchronous sub-vehicles**.
- **Nested MultiVehicles** — a MultiVehicle as a sub-vehicle of another.
- **Conduits with all optional modules** — FeeManager, AccountList, OwnerRegistry.
- **Fork tests** against live deployments on Ethereum Mainnet and Base.

7.4 Automatic allocations to vehicles with deposit fees break share fairness: Proof of Concept

```
export MAINNET_BLOCK_NUMBER=25071042
export MAINNET_RPC=YOUR_RPC
forge test --mc MultilevelRailnet_MainnetFork --mt test_DepositFeesBreakShareFairnessMVOnly
↪  -vvv
```

The first test file: `MultilevelRailnet.fork.t.sol`.

```
//
// SPDX-License-Identifier: BUSL-1.1
//
//                    ▒▒ ▒▒                    ▒▒
//                       ▒▒                    ▒▒
//    ▒▒ ▒▒▒  ▒▒▒▒▒  ▒▒ ▒▒ ▒▒▒▒▒▒  ▒▒▒▒▒▒  ▒▒▒▒▒▒
//    ▒▒▒    ▒▒   ▒▒ ▒▒ ▒▒ ▒▒   ▒▒ ▒▒   ▒▒ ▒▒
//    ▒▒     ▒▒▒▒▒▒▒ ▒▒ ▒▒ ▒▒   ▒▒ ▒▒▒▒▒▒▒ ▒▒
//    ▒▒     ▒▒      ▒▒ ▒▒ ▒▒   ▒▒ ▒▒      ▒▒
//    ▒▒      ▒▒▒▒▒▒ ▒▒ ▒▒ ▒▒   ▒▒  ▒▒▒▒▒▒  ▒▒▒▒
//
//    ================================================
//
pragma solidity >=0.8.33;

// solhint-disable contract-name-capwords
// solhint-disable no-console

import {console} from "@forge-std/console.sol";
import {IERC20} from "@openzeppelin-contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from
↪  "@openzeppelin-contracts/token/ERC20/extensions/IERC20Metadata.sol";

import {Asset, Mode, Query, EstimationType} from "src/steam/Query.sol";
import {IVehicle} from "src/steam/IVehicle.sol";
import {Conduit} from "src/conduit/Conduit.sol";
import {ConduitStructs} from "src/conduit/libs/ConduitStructs.sol";
import {IAccountList} from "src/conduit/interfaces/IAccountList.sol";
import {IOwnerRegistry} from "src/conduit/interfaces/IOwnerRegistry.sol";
import {ConduitFactory} from "src/factories/conduit/ConduitFactory.sol";
import {IBaseVehicle} from "src/vehicles/base/interfaces/IBaseVehicle.sol";
import {IFeeManager} from "src/vehicles/base/interfaces/IFeeManager.sol";

import {ForkUtils} from "test/test_utils/ForkUtils.sol";
import {MultilevelRailnetForkConfig} from "../MultilevelRailnetForkConfig.sol";
```

```
/// @title MultilevelRailnet_MainnetFork
/// @notice Smoke test that verifies the multilevel Railnet stack (Aave + Morpho vehicles ⊂
↪  MultiVehicle ⊂ Conduit)
///         can be deployed against a real mainnet fork and that a user can:
///         1. Deposit USDC into the Conduit and receive conduit shares.
///         2. See the funds dispatched into both Aave and Morpho through the MultiVehicle's
↪  strategy queue.
///         3. Observe fee accounting (10% perf / 1% mgmt) reflected on every layer that owns a
↪  FeeManager.
contract MultilevelRailnet_MainnetFork is MultilevelRailnetForkConfig, ForkUtils {
    function setUp() public {
        _startFork("MAINNET");
        _deployAll();
    }

    function test_DepositFeesBreakShareFairnessMVOnly() public {
        $aaveFeeManager.setFees(IFeeManager.Fees({
            performanceFeeBps: $PERFORMANCE_FEE_BPS,
            managementFeeBps: $MANAGEMENT_FEE_BPS,
            depositFeeBps: $MAX_DEPOSIT_FEE_BPS,// @audit-info set fees to 10% on deposits
            redeemFeeBps: $REDEEM_FEE_BPS
        }));
        console.log("--------------------    Initial state     --------------------");
        console.log("MV      total assets:  ", $multiVehicle.totalAssets());
        console.log("Aave    total assets:  ", $aaveVehicle.totalAssets());
        console.log("Morpho  total assets:  ", $morphoVehicle.totalAssets());


        console.log("\n");
        console.log("MV fee manager address:  ", address($multiVehicle.feeManager()));
        console.log("Fee Mgr balance in MV :  ",
        ↪  $multiVehicle.balanceOf(address($multiVehicle.feeManager())));


        console.log("\n--------------------    User1: Deposit 10k USDC
        ↪  --------------------");
        address _user = makeAddr("ml-user");
        uint256 _amount = 10_000 * 10 ** $usdc.decimals(); // 10_000 USDC

        deal(address($usdc), _user, _amount);

        // 1. User deposits into the Conduit.
        Asset[] memory _input = new Asset[](1);
        _input[0] = Asset({asset: address($usdc), value: _amount});
        Asset[] memory _output = $multiVehicle.estimate(_input, Mode.DEPOSIT,
        ↪  EstimationType.OUTPUT);

        Query memory _depositQuery = Query({
            owner: address(_user),
            receiver: address(_user),
            input: _input,
            output: _output,
            mode: Mode.DEPOSIT,
            salt: keccak256("ml-deposit-1"),
            data: ""
        });

        vm.startPrank(_user);
        IERC20($usdc).approve(address($multiVehicle), _amount);
        $multiVehicle.create(_depositQuery);
        $multiVehicle.unlock(_depositQuery);
        vm.stopPrank();
```

```
console.log("MV      total assets:  ", $multiVehicle.totalAssets());
console.log("Aave    total assets:  ", $aaveVehicle.totalAssets());
console.log("Morpho  total assets:  ", $morphoVehicle.totalAssets());


console.log("\n");
console.log("User shares in MV:  ",
↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user));

Asset[] memory estAsset = new Asset[](1);
estAsset[0] = Asset({asset: address($multiVehicle), value:
↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user)});
Asset[] memory _redeemEstimate = $multiVehicle.estimate(estAsset, Mode.REDEEM,
↪  EstimationType.OUTPUT);
console.log("Estimated redeem output: ", _redeemEstimate[0].value);


console.log("\n--------------------    User2: Deposit and withdraw 10k USDC
↪  --------------------");
address _user2 = makeAddr("ml-user-2");

deal(address($usdc), _user2, _amount);
// 1. User 2 deposits into the Conduit.
_input = new Asset[](1);
_input[0] = Asset({asset: address($usdc), value: $usdc.balanceOf(_user2)});
_output = $multiVehicle.estimate(_input, Mode.DEPOSIT, EstimationType.OUTPUT);

_depositQuery = Query({
    owner: address(_user2),
    receiver: address(_user2),
    input: _input,
    output: _output,
    mode: Mode.DEPOSIT,
    salt: keccak256("ml-deposit-1"),
    data: ""
});

vm.startPrank(_user2);
IERC20($usdc).approve(address($multiVehicle), _amount);
$multiVehicle.create(_depositQuery);
$multiVehicle.unlock(_depositQuery);
vm.stopPrank();

console.log("User1 shares in Conduit:  ",
↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user));
console.log("User2 shares in Conduit:  ",
↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user2));
estAsset = new Asset[](1);
estAsset[0] = Asset({asset: address($multiVehicle), value:
↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user)});
_redeemEstimate = $multiVehicle.estimate(estAsset, Mode.REDEEM, EstimationType.OUTPUT);
console.log("User 1 - Estimated redeem output: ", _redeemEstimate[0].value);

_input[0] = Asset({asset: address($multiVehicle), value:
↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user2)});
_output = $multiVehicle.estimate(_input, Mode.REDEEM, EstimationType.OUTPUT);
console.log("User 2 - Estimated redeem output: ", _output[0].value);

Query memory _redeemQuery = Query({
    owner: address(_user2),
    receiver: address(_user2),
    input: _input,
```

```
        output: _output,
        mode: Mode.REDEEM,
        salt: keccak256("ml-redeem-1"),
        data: ""
    });

    vm.startPrank(_user2);
    IERC20(address($multiVehicle)).approve(address($multiVehicle),
    ↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user2));
    $multiVehicle.create(_redeemQuery);
    $multiVehicle.unlock(_redeemQuery);
    vm.stopPrank();
    console.log("MV      total assets:  ", $multiVehicle.totalAssets());
    console.log("Aave    total assets:  ", $aaveVehicle.totalAssets());
    console.log("Morpho  total assets:  ", $morphoVehicle.totalAssets());


    console.log("\n");
    console.log("User2 shares in Conduit:  ",
    ↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user2));


    console.log("\n--------------------    Final situation     --------------------");

    console.log("User1 shares in Conduit:  ",
    ↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user));
    console.log("User2 shares in Conduit:  ",
    ↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user2));

    _input[0] = Asset({asset: address($multiVehicle), value:
    ↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user)});
    _redeemEstimate = $multiVehicle.estimate(_input, Mode.REDEEM, EstimationType.OUTPUT);
    console.log("User 1 - Estimated redeem output: ", _redeemEstimate[0].value);
    _redeemQuery = Query({
        owner: address(_user),
        receiver: address(_user),
        input: _input,
        output: _redeemEstimate,
        mode: Mode.REDEEM,
        salt: keccak256("ml-redeem-1"),
        data: ""
    });

    vm.startPrank(_user);
    IERC20(address($multiVehicle)).approve(address($multiVehicle),
    ↪  IERC20Metadata(address($multiVehicle)).balanceOf(_user));
    $multiVehicle.create(_redeemQuery);
    $multiVehicle.unlock(_redeemQuery);
    vm.stopPrank();
    console.log("MV      total assets:  ", $multiVehicle.totalAssets());
    console.log("Aave    total assets:  ", $aaveVehicle.totalAssets());
    console.log("Morpho  total assets:  ", $morphoVehicle.totalAssets());
    console.log("\n");
    console.log("User1 USDC balance:  ", $usdc.balanceOf(_user));
    console.log("User2 USDC balance:  ", $usdc.balanceOf(_user2));
}



/* ------------------------------------------------------------------------------------------
↪  -------------------------------- */
```

```
/*                                                                HELPERS
↪  */
/* ---------------------------------------------------------------------------------------
↪  ---------------------------------- */

function _assertFees(IFeeManager feeManager, string memory label) internal view {
    (, IFeeManager.Fees memory _fees) = feeManager.fees();
    assertEq(_fees.performanceFeeBps, $PERFORMANCE_FEE_BPS, string.concat(label, ":
    ↪  performanceFeeBps"));
    assertEq(_fees.managementFeeBps, $MANAGEMENT_FEE_BPS, string.concat(label, ":
    ↪  managementFeeBps"));
    assertEq(_fees.depositFeeBps, $DEPOSIT_FEE_BPS, string.concat(label, ":
    ↪  depositFeeBps"));
    assertEq(_fees.redeemFeeBps, $REDEEM_FEE_BPS, string.concat(label, ": redeemFeeBps"));
}

/// @dev Initial deposit performed by `MultiVehicleFactory.spawn` (1 USDC). Captured at
↪  first call to keep
///      reuse cheap across multiple assertions in the same test.
function _initialMultiVehicleAssets() internal view returns (uint256) {
    return 10 ** $usdc.decimals();
}

function _setPerformanceOnlyFees(IFeeManager feeManager) internal {
    feeManager.setFees(IFeeManager.Fees({
        performanceFeeBps: $PERFORMANCE_FEE_BPS,
        managementFeeBps: 0,
        depositFeeBps: $DEPOSIT_FEE_BPS,
        redeemFeeBps: $REDEEM_FEE_BPS
    }));
}

function _spawnFeeEnabledConduit(IFeeManager feeManager) internal returns (Conduit
↪  conduit) {
    ConduitFactory.SpawnParams memory _params = ConduitFactory.SpawnParams({
        name: "Performance Fee Conduit",
        symbol: "PFC",
        vehicle: IVehicle(address($multiVehicle)),
        feeManager: feeManager,
        accountList: IAccountList(address(0)),
        ownerRegistry: IOwnerRegistry(address(0)),
        accessControl: $sharedAC,
        transferMode: ConduitStructs.TransferMode.ALLOW_TRANSFER,
        initialExpectedSupply: 1,
        querySalt: keccak256("multilevel-conduit-fee-query"),
        deploymentSalt: keccak256("multilevel-conduit-fee-deployment")
    });

    uint256 _initialDeposit =
    ↪  $conduitFactory.ASSET_REGISTRY().getInitialDepositAmount(address($usdc));
    deal(address($usdc), address(this), _initialDeposit);
    IERC20(address($usdc)).approve(address($conduitFactory), 0);
    IERC20(address($usdc)).approve(address($conduitFactory), _initialDeposit);

    conduit = Conduit(address($conduitFactory.spawn(_params, _params.deploymentSalt)));
}

function _estimateConduitRedeem(Conduit conduit, uint256 conduitShares) internal view
↪  returns (uint256) {
    Asset[] memory _input = new Asset[](1);
    _input[0] = Asset({asset: address(conduit), value: conduitShares});
    Asset[] memory _output = conduit.estimate(_input, Mode.REDEEM, EstimationType.OUTPUT);
```

```
        return _output[0].value;
    }

}
```

The second file: `MultilevelRailnetForkConfig.sol`.

```
// SPDX-License-Identifier: BUSL-1.1
//
//
//
//
//
//
//
//
//
//
pragma solidity >=0.8.33;

// solhint-disable kiln-rules/ordering
// solhint-disable no-console

import {Test, console} from "@forge-std/Test.sol";
import {IERC20} from "@openzeppelin-contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from
↪  "@openzeppelin-contracts/token/ERC20/extensions/IERC20Metadata.sol";
import {SafeERC20} from "@openzeppelin-contracts/token/ERC20/utils/SafeERC20.sol";
import {Math} from "@openzeppelin-contracts/utils/math/Math.sol";

import {ICreateX} from "@createx/ICreateX.sol";
import {BeaconProxy} from "@openzeppelin-contracts/proxy/beacon/BeaconProxy.sol";
import {IMorpho, Id as MarketId, MarketParams} from "@morpho/interfaces/IMorpho.sol";

// Railnet - common
import {Asset, EstimationType, Mode, Query, State} from "src/steam/Query.sol";
import {ExternalAccessControl} from "src/common/ExternalAccessControl.sol";
import {IExternalAccessControl} from "src/common/interfaces/IExternalAccessControl.sol";
import {FreezablePausableBeacon} from "src/common/FreezablePausableBeacon.sol";
import {SharesLib} from "src/libs/Shares.sol";
import {Roles} from "src/libs/Roles.sol";

// Railnet - factories
import {AccessControlFactory} from "src/factories/common/AccessControlFactory.sol";
import {AaveV3VehicleFactory} from "src/factories/vehicles/AaveV3VehicleFactory.sol";
import {ConduitFactory} from "src/factories/conduit/ConduitFactory.sol";
import {CoreFactory} from "src/factories/CoreFactory.sol";
import {FeeManagerFactory} from "src/factories/vehicles/FeeManagerFactory.sol";
import {ModulesManagerFactory} from "src/factories/vehicles/ModulesManagerFactory.sol";
import {MorphoBlueVehicleFactory} from "src/factories/vehicles/MorphoBlueVehicleFactory.sol";
import {MultiVehicleFactory} from "src/factories/vehicles/MultiVehicleFactory.sol";

// Railnet - vehicles
import {AaveV3Vehicle} from "src/vehicles/aave_v3/AaveV3Vehicle.sol";
import {IPool} from "src/vehicles/aave_v3/interfaces/IPool.sol";
import {IPoolAddressesProvider} from
↪  "src/vehicles/aave_v3/interfaces/IPoolAddressesProvider.sol";
import {IPoolDataProvider} from "src/vehicles/aave_v3/interfaces/IPoolDataProvider.sol";
import {TokenMath} from "@aave-v3-libs/helpers/TokenMath.sol";
import {Interceptor} from "src/abstracts/Interceptor.sol";
```

```
import {FeeManager} from "src/vehicles/base/FeeManager.sol";
import {ModulesManager} from "src/vehicles/base/ModulesManager.sol";
import {IBaseVehicle} from "src/vehicles/base/interfaces/IBaseVehicle.sol";
import {IFeeManager} from "src/vehicles/base/interfaces/IFeeManager.sol";
import {IVehicle} from "src/steam/IVehicle.sol";
import {MorphoBlueVehicle} from "src/vehicles/morpho_blue/MorphoBlueVehicle.sol";
import {MorphoBlueVehicleStructs} from
↪  "src/vehicles/morpho_blue/libs/MorphoBlueVehicleStructs.sol";
import {MultiVehicle} from "src/vehicles/multi/MultiVehicle.sol";
import {QueryRedeemQueue} from "src/vehicles/multi/QueryRedeemQueue.sol";
import {QueueStrategyEngine} from "src/vehicles/multi/QueueStrategyEngine.sol";
import {SectorAccountingEngine} from "src/vehicles/multi/SectorAccountingEngine.sol";
import {SubQueryEngine} from "src/vehicles/multi/SubQueryEngine.sol";
import {VehicleRegistry} from "src/vehicles/multi/VehicleRegistry.sol";
import {Target} from "src/vehicles/multi/libs/Target.sol";
import {IQueueStrategyEngine} from "src/vehicles/multi/interfaces/IQueueStrategyEngine.sol";

// Railnet - conduit
import {AccountList} from "src/conduit/AccountList.sol";
import {Conduit} from "src/conduit/Conduit.sol";
import {ConduitStructs} from "src/conduit/libs/ConduitStructs.sol";
import {OwnerRegistry} from "src/conduit/OwnerRegistry.sol";
import {IAccountList} from "src/conduit/interfaces/IAccountList.sol";
import {IConduit} from "src/conduit/interfaces/IConduit.sol";
import {IOwnerRegistry} from "src/conduit/interfaces/IOwnerRegistry.sol";

// Test utils
import {Deployer} from "test/test_utils/Deployer.sol";

/// @title Multilevel Railnet Fork Config
/// @notice Reusable scaffolding that wires three Railnet layers together against a real
↪  network:
///         - Level 0: AAVE V3 vehicle (USDC) and Morpho Blue vehicle (USDC market).
///         - Level 1: a MultiVehicle (USDC) that allocates between Aave and Morpho.
///         - Level 2: a Conduit linked to the MultiVehicle.
///         All Level 0 and Level 1 vehicles are configured with 10% performance fee, 1%
↪  management fee,
///         and 0% deposit / redeem fees.
contract MultilevelRailnetForkConfig is Test, Deployer {
    using SafeERC20 for IERC20;

    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */
    /*                                                                  NETWORK
    ↪  */
    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */

    /// @dev Mainnet USDC.
    address internal constant $USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    /// @dev Aave V3 PoolAddressesProvider (mainnet).
    address internal constant $AAVE_POOL_ADDRESSES_PROVIDER =
    ↪  0x2f39d218133AFaB8F2B819B1066c7E434Ad94E9e;
    /// @dev Morpho Blue (deployed at the same address on every chain).
    address internal constant $MORPHO = 0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb;
    /// @dev Morpho Blue USDC / cbBTC market id (mainnet).
    MarketId internal constant $MORPHO_USDC_MARKET =
        MarketId.wrap(0x64d65c9a2d91c36d56fbc42d69e979335320169b3df63bf92789e2c8883fcc64);

    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */
```

```
/*                                                          FEE PARAMETERS
↪  */
/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */

/// @dev 10% performance fee.
uint32 internal constant $PERFORMANCE_FEE_BPS = 1000;
/// @dev 1% management fee.
uint32 internal constant $MANAGEMENT_FEE_BPS = 100;
/// @dev No deposit fee.
uint32 internal constant $DEPOSIT_FEE_BPS = 0;
uint32 internal constant $MAX_DEPOSIT_FEE_BPS = 1000;
/// @dev No redeem fee.
uint32 internal constant $REDEEM_FEE_BPS = 0;
uint32 internal constant $MAX_REDEEM_FEE_BPS = 1000;

/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */
/*                                                          DEPLOYED STATE
↪  */
/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */

address public $deployer;
address public $feeRecipient;

IERC20Metadata public $usdc;

CoreFactory public $coreFactoryRef;
ExternalAccessControl public $sharedAC;

// Level 0 - vehicles
AaveV3Vehicle public $aaveVehicle;
MorphoBlueVehicle public $morphoVehicle;
FeeManager public $aaveFeeManager;
FeeManager public $morphoFeeManager;

// Level 1 - multi vehicle
MultiVehicle public $multiVehicle;
QueueStrategyEngine public $strategyEngine;
SectorAccountingEngine public $accountingEngine;
QueryRedeemQueue public $redeemQueue;
SubQueryEngine public $subQueryEngine;
VehicleRegistry public $vehicleRegistry;
FeeManager public $multiFeeManager;

// Level 2 - conduit
Conduit public $conduit;
ConduitFactory public $conduitFactory;
FreezablePausableBeacon public $conduitBeacon;

/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */
/*                                                                DEPLOY
↪  */
/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */

/// @notice Deploys every level. Must be invoked from a `setUp()` after the fork is
↪  selected.
function _deployAll() internal {
    $deployer = makeAddr("multilevel-deployer");
```

```
        $feeRecipient = makeAddr("multilevel-fee-recipient");
        $usdc = IERC20Metadata($USDC);

        $coreFactoryRef = _deployCoreFactory();
        $sharedAC = _deployAccessControlIfNeeded(
            $coreFactoryRef,
            AccessControlFactory.SpawnParams({
                initialDelay: 0,
                initialDefaultAdmin: address(this),
                initialRoles: new IExternalAccessControl.RoleAttribution[](0),
                deploymentSalt: keccak256("multilevel-shared-ac")
            })
        );

        // Make STEAM deposit/redeem permissionless so the SubQueryEngine can interact with
        ↪  sub-vehicles
        // and the Conduit can interact with the MultiVehicle. This mirrors the public role
        ↪  setup used
        // by `MultiVehicleTester` and avoids granting the role per address.
        $sharedAC.setRolePublic(Roles.VEHICLE_STEAM_DEPOSIT, true);
        $sharedAC.setRolePublic(Roles.VEHICLE_STEAM_REDEEM, true);

        // Register USDC in the asset registry with the canonical 1 USDC initial deposit
        ↪  amount.
        _registerAsset(_getOrDeployAssetRegistry(), $USDC);

        _deployLevel0();
        _deployLevel1();
        _deployLevel2();
    }

    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */
    /*                                                                 LEVEL 0
    ↪  */
    /* ---------------------------------------------------------------------------------------
    ↪  --------------------------------- */

    function _deployLevel0() internal {
        // Per-vehicle fee managers - each gets the same 10% perf / 1% mgmt / 0/0 transactional
        ↪  split.
        $aaveFeeManager = _spawnFeeManager(keccak256("multilevel-fee-manager-aave"));
        $morphoFeeManager = _spawnFeeManager(keccak256("multilevel-fee-manager-morpho"));

        // AAVE V3 vehicle (USDC).
        AaveV3VehicleInfra memory _aaveInfra = _deployAaveV3VehicleInfra($deployer,
        ↪  $coreFactoryRef);
        uint256 _aaveExpectedSupply = _previewAaveInitialSupply($USDC);
        $aaveVehicle = _deployAaveV3VehicleInstance(
            $deployer,
            AaveV3VehicleFactory.SpawnParams({
                asset: $USDC,
                poolAddressesProvider: $AAVE_POOL_ADDRESSES_PROVIDER,
                accessControl: $sharedAC,
                feeManager: $aaveFeeManager,
                modulesManager: ModulesManager(address(0)),
                querySalt: keccak256("multilevel-aave-query"),
                deploymentSalt: keccak256("multilevel-aave-deployment"),
                forbiddenAddresses: new address[](0),
                initialExpectedSupply: _aaveExpectedSupply
            }),
            _aaveInfra
```

```
    );

    // Morpho Blue vehicle (USDC market).
    MorphoBlueVehicleInfra memory _morphoInfra = _deployMorphoBlueVehicleInfra(
        $deployer, MorphoBlueVehicleStructs.ImmutableParams({morpho: IMorpho($MORPHO)}),
        ↪  $coreFactoryRef
    );
    uint256 _morphoExpectedSupply = _previewMorphoInitialSupply($MORPHO_USDC_MARKET);
    $morphoVehicle = _deployMorphoBlueVehicleInstance(
        $deployer,
        MorphoBlueVehicleFactory.SpawnParams({
            morpho: $MORPHO,
            marketId: $MORPHO_USDC_MARKET,
            accessControl: $sharedAC,
            feeManager: $morphoFeeManager,
            modulesManager: ModulesManager(address(0)),
            querySalt: keccak256("multilevel-morpho-query"),
            deploymentSalt: keccak256("multilevel-morpho-deployment"),
            forbiddenAddresses: new address[](0),
            initialExpectedSupply: _morphoExpectedSupply
        }),
        _morphoInfra
    );
}

/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */
/*                                                        LEVEL 1
↪  */
/* ------------------------------------------------------------------------------------------
↪  ---------------------------------- */

function _deployLevel1() internal {
    $multiFeeManager = _spawnFeeManager(keccak256("multilevel-fee-manager-multi"));

    MultiVehicleInfra memory _multiInfra = _deployMultiVehicleInfra($deployer,
    ↪  $coreFactoryRef);

    MultiVehicleFactory.SpawnParams memory _params = MultiVehicleFactory.SpawnParams({
        asset: $usdc,
        name: "Multilevel",
        symbol: "ML",
        initialInterceptions: new Interceptor.Interception[](0),
        accessControl: $sharedAC,
        feeManager: $multiFeeManager,
        modulesManager: ModulesManager(address(0)),
        salts: MultiVehicleFactory.Salts({
            multiVehicle: keccak256("multilevel-multi-multiVehicle"),
            queryRedeemQueue: keccak256("multilevel-multi-queryRedeemQueue"),
            queueStrategyEngine: keccak256("multilevel-multi-queueStrategyEngine"),
            sectorAccountingEngine: keccak256("multilevel-multi-sectorAccountingEngine"),
            subQueryEngine: keccak256("multilevel-multi-subQueryEngine"),
            vehicleRegistry: keccak256("multilevel-multi-vehicleRegistry"),
            initialDepositQuery: keccak256("multilevel-multi-initialDepositQuery")
        }),
        forbiddenAddresses: new address[](0)
    });

    MultiVehicleFactory.Contracts memory _contracts =
    ↪  _deployMultiVehicleInstance($deployer, _params, _multiInfra);
    $multiVehicle = MultiVehicle(_contracts.multiVehicle);
    $strategyEngine = QueueStrategyEngine(_contracts.queueStrategyEngine);
```

```
        $accountingEngine = SectorAccountingEngine(_contracts.sectorAccountingEngine);
        $redeemQueue = QueryRedeemQueue(_contracts.queryRedeemQueue);
        $subQueryEngine = SubQueryEngine(_contracts.subQueryEngine);
        $vehicleRegistry = VehicleRegistry(_contracts.vehicleRegistry);

        // Authorize both sub-vehicles and configure a 50/50 deposit queue.
        // The MultiVehicle's own access control is `$sharedAC`; we already hold
        ↪  DEFAULT_ADMIN_ROLE there.
        $sharedAC.grantRole(Roles.MULTI_VEHICLE_SET_VEHICLE_AUTHORIZATION, address(this));
        $sharedAC.grantRole(Roles.MULTI_VEHICLE_SET_QUEUES, address(this));
        $sharedAC.grantRole(Roles.MULTI_VEHICLE_DISPATCH, address(this));
        $sharedAC.grantRole(Roles.MULTI_VEHICLE_REBALANCE, address(this));
        $sharedAC.grantRole(Roles.MULTI_VEHICLE_FEED_QUERY_REDEEM_QUEUE, address(this));
        $sharedAC.grantRole(Roles.MULTI_VEHICLE_RETRIEVE_QUERY_REDEEM_QUEUE_ASSETS,
        ↪  address(this));
        $sharedAC.grantRole(Roles.MULTI_VEHICLE_PROGRESS_QUERY, address(this));
        $sharedAC.grantRole(Roles.FEE_MANAGER_SET_FEES, address(this));

        $vehicleRegistry.authorize(IVehicle(address($aaveVehicle)));
        $vehicleRegistry.authorize(IVehicle(address($morphoVehicle)));

        IQueueStrategyEngine.QueueEntry[] memory _depositQueue = new
        ↪  IQueueStrategyEngine.QueueEntry[](2);
        _depositQueue[0] = IQueueStrategyEngine.QueueEntry({
            vehicle: IBaseVehicle(address($aaveVehicle)), target: Target({value:
            ↪  type(uint256).max, threshold: 0})
        });
        _depositQueue[1] = IQueueStrategyEngine.QueueEntry({
            vehicle: IBaseVehicle(address($morphoVehicle)), target: Target({value:
            ↪  type(uint256).max, threshold: 0})
        });

        IQueueStrategyEngine.QueueEntry[] memory _redeemQueue = new
        ↪  IQueueStrategyEngine.QueueEntry[](2);
        _redeemQueue[1] = IQueueStrategyEngine.QueueEntry({
            vehicle: IBaseVehicle(address($aaveVehicle)), target: Target({value: 0, threshold:
            ↪  0})
        });
        _redeemQueue[0] = IQueueStrategyEngine.QueueEntry({
            vehicle: IBaseVehicle(address($morphoVehicle)), target: Target({value: 0,
            ↪  threshold: 0})
        });
        $strategyEngine.setQueues(_depositQueue, _redeemQueue);
    }

    /* ------------------------------------------------------------------------------------------
    ↪  ---------------------------------- */
    /*                                                        LEVEL 2
    ↪  */
    /* ------------------------------------------------------------------------------------------
    ↪  ---------------------------------- */

    function _deployLevel2() internal {
        Conduit _impl = new Conduit();
        $conduitBeacon = new FreezablePausableBeacon(address(_impl), $sharedAC);
        $conduitFactory = new ConduitFactory($coreFactoryRef, $conduitBeacon, $sharedAC,
        ↪  _getOrDeployAssetRegistry());

        $sharedAC.grantRole(Roles.CONDUIT_SPAWN, address(this));

        // The conduit has no fee manager nor account list - the prompt only requested fees on
        ↪  the lower
```

```
        // levels. Transfer mode is permissionless so users can move conduit shares freely.
        ConduitFactory.SpawnParams memory _params = ConduitFactory.SpawnParams({
            name: "Multilevel Conduit",
            symbol: "ML-CDT",
            vehicle: IVehicle(address($multiVehicle)),
            feeManager: IFeeManager(address(0)),
            accountList: IAccountList(address(0)),
            ownerRegistry: IOwnerRegistry(address(0)),
            accessControl: $sharedAC,
            transferMode: ConduitStructs.TransferMode.ALLOW_TRANSFER,
            initialExpectedSupply: 1,
            querySalt: keccak256("multilevel-conduit-query"),
            deploymentSalt: keccak256("multilevel-conduit-deployment")
        });

        // Pre-fund the initial deposit and approve the factory.
        uint256 _initialDeposit =
         ↪  $conduitFactory.ASSET_REGISTRY().getInitialDepositAmount($USDC);
        console.log("Initial deposit amount: ", _initialDeposit);
        deal($USDC, address(this), _initialDeposit);
        IERC20($USDC).forceApprove(address($conduitFactory), _initialDeposit);

        $conduit = Conduit(address($conduitFactory.spawn(_params, _params.deploymentSalt)));
    }

    /* ---------------------------------------------------------------------------------------------
    ↪  --------------------------------- */
    /*                                                                HELPERS
    ↪  */
    /* ---------------------------------------------------------------------------------------------
    ↪  --------------------------------- */

    /// @dev Spawns a FeeManager with the multilevel fee profile (10% perf, 1% mgmt, 0/0
    ↪  transactional).
    function _spawnFeeManager(bytes32 salt) internal returns (FeeManager feeManager) {
        FeeManagerInfra memory _infra = _deployFeeManagerInfra($deployer, $coreFactoryRef,
        ↪  false);

        IFeeManager.FeeRecipient[] memory _recipients = new IFeeManager.FeeRecipient[](1);
        _recipients[0] = IFeeManager.FeeRecipient({target: $feeRecipient, shareBps: 10_000});

        FeeManagerFactory.SpawnParams memory _params = FeeManagerFactory.SpawnParams({
            accessControl: $sharedAC,
            initialFees: IFeeManager.Fees({
                performanceFeeBps: $PERFORMANCE_FEE_BPS,
                managementFeeBps: $MANAGEMENT_FEE_BPS,
                depositFeeBps: $DEPOSIT_FEE_BPS,
                redeemFeeBps: $REDEEM_FEE_BPS
            }),
            initialMaxFees: IFeeManager.Fees({
                performanceFeeBps: $PERFORMANCE_FEE_BPS,
                managementFeeBps: $MANAGEMENT_FEE_BPS,
                depositFeeBps: $MAX_DEPOSIT_FEE_BPS,
                redeemFeeBps: $MAX_REDEEM_FEE_BPS
            }),
            initialRecipients: _recipients,
            deploymentSalt: salt
        });

        feeManager = _deployFeeManagerInstance($deployer, _params, _infra);
    }
```

```
    /// @dev Replicates `AaveV3VehicleForkConfig._deployVehicle` initial-supply math so the
    ↪  factory's
    ///      slippage check `initialExpectedSupply` succeeds.
    function _previewAaveInitialSupply(address asset) internal view returns (uint256) {
        uint8 _assetDecimals = IERC20Metadata(asset).decimals();
        address _pool = IPoolAddressesProvider($AAVE_POOL_ADDRESSES_PROVIDER).getPool();
        uint256 _liquidityIndex = IPool(_pool).getReserveNormalizedIncome(asset);
        uint256 _initialDepositAmount = 10 ** uint256(_assetDecimals);
        uint256 _expectedScaledShares =
        ↪  TokenMath.getATokenMintScaledAmount(_initialDepositAmount, _liquidityIndex);
        return SharesLib.scale(_expectedScaledShares, _assetDecimals, 18, Math.Rounding.Floor);
    }

    /// @dev Replicates `MorphoBlueVehicleForkConfig._deployVehicle` initial-supply math:
    ↪  simulate a 1-token
    ///      supply against the live market, take the resulting supplyShares delta, and scale
    ↪  to 18 decimals.
    function _previewMorphoInitialSupply(MarketId marketId) internal returns (uint256) {
        MarketParams memory _params = IMorpho($MORPHO).idToMarketParams(marketId);
        uint8 _assetDecimals = IERC20Metadata(_params.loanToken).decimals();
        uint256 _initialDepositAmount = 10 ** uint256(_assetDecimals);

        uint256 _snap = vm.snapshotState();
        deal(_params.loanToken, address(this), _initialDepositAmount);
        IERC20(_params.loanToken).forceApprove($MORPHO, _initialDepositAmount);
        uint256 _supplySharesBefore = IMorpho($MORPHO).position(marketId,
        ↪  address(this)).supplyShares;
        IMorpho($MORPHO).supply(_params, _initialDepositAmount, 0, address(this), hex"");
        uint256 _morphoSharesReceived =
            IMorpho($MORPHO).position(marketId, address(this)).supplyShares -
            ↪  _supplySharesBefore;
        vm.revertToState(_snap);

        return SharesLib.scale(_morphoSharesReceived, _assetDecimals + 6, 18,
        ↪  Math.Rounding.Floor);
    }
}
```