# Table of contents

# Intro

Enzyme Protocol is an Ethereum-based protocol for decentralized on-chain asset management. It is a protocol for people or entities to manage their wealth & the wealth of others within a customizable and safe environment. Enzyme empowers anyone to set up, manage and invest in customized on-chain investment vehicles. ****

The purpose of this document is to provide a high-level overview of the aims, architectural decisions, governance, GitHub repo, and other idiosyncrasies of the "v2" protocol release.

# GitHub repo

**All code for audit is located in the `develop` branch of our private repo: https://github.com/avantgardefinance/protocol**

### Contracts

All contracts used for both production and testing are located in `contracts/`

`mocks/` - (Not for audit) Contains mock contracts used by tests and for our own testing environment

`persistent/` - Production contracts that will not be changed after this release. These are contracts for scaffolding the overarching structure that facilitates fund migration from an old release to the current release.

`release/` - Production contracts that will be deployed for this release, and will likely change (and be re-deployed) for subsequent releases.

### Tests

All tests are in the `tests/` directory.

Rather than rigidly segregating traditional "unit", "integration", and "end-to-end" tests, we blend "unit" and "integration" to test our own internal state transitions, inputs, and outputs, and only use "end-to-end" tests for testing that our interactions with external protocols and contracts works as expected.

End-to-end tests are found in `release/e2e/`

All other tests are located in directories and files corresponding to their contract name and roughly corresponding with the file path of the primary contract being tested.

# Persistent Architecture

### Philosophy

One of the biggest goals for Enzyme v2 is to have upgradable funds. Finding an appropriate pattern was a challenging task because:

- We have different types of contracts that comprise and are shared by funds, e.g., fund libraries, plugins (fees, policies, and DeFi adapters), infrastructure that can persist between releases, etc,
- It is important to our philosophy for both fund managers and investors to have the opportunity to opt-in or opt-out of every update (i.e., we can't change how a PerformanceFee works for every fund that is already using it)

### Approach

The pattern that we settled on is a user-initiated partial migration (essential state only) from an old release to the current release.

The essential state for a fund is:

- its holdings (i.e., token balances)
- its accounting of investor shares in the fund
- access control for ownership and state-changing interactions

This migration pattern is accomplished through two persistent contracts: a `Dispatcher` and per-fund `VaultProxy` instances.

**VaultProxy** The "essential state" described above lives in per-fund `VaultProxy` contract instances, which are upgradable contracts following the EIP-1822 proxy pattern.

The `VaultProxy` specifies a `VaultLib` as its target logic, and these logic contracts are deployed per release, and swapped out during a migration.

A `VaultLibBaseCore` contract defines the absolutely essential state and logic for every VaultProxy. This includes:

- a standard ERC20 implementation called `SharesTokenBase`
- the functions required of a `IProxiableVault` called by the `Dispatcher`
- core access control roles: `owner`, `accessor`, `creator`, and `migrator`

The `owner` is the fund's owner.

The `migrator` is an optional role to allow a non-owner to migrate the fund

The `creator` is the `Dispatcher` contract, and only this role is allowed to update the `accessor` and `vaultLib` values.

The `accessor` is the primary account that can make state-changing calls to the `VaultProxy` . In practice, this is the release-level contract that interacts with a vault's assets, updates shares balances, etc.

This extremely abstract interface - in which a `VaultProxy` needs no knowledge about a release other than which caller can write state - allows for nearly limitless possibilities for release-level architecture.

The `VaultLibBaseCore` contract can be extended with new storage and events by implementing a new `VaultLibBaseN` that extends the previous base. For example, this first release implements `abstract contract VaultLibBase1 is VaultLibBaseCore` . The next release that adds to state or events would continue this pattern with `abstract contract VaultLibBase2 is VaultLibBase1` , and so on.

The release itself will provide a `VaultLib` that extends the latest `VaultLibBaseN` implementation with the required logic of that release.

**Dispatcher** An overarching, non-upgradable `Dispatcher` contract is charged with:

- deploying new instances of `VaultProxy`
- migrating a `VaultProxy` from an old release to the current release
- maintaining global state such as the current release, the global owner (i.e., the Enzyme Council) and the default `symbol` value for fund shares tokens

The `Dispatcher` stores the `currentFundDeployer` (a generic reference to the latest release's contract that is responsible for deploying and migrating funds), and only a `msg.sender` with that value is allowed to call functions to deploy or migrate a `VaultProxy` .

This release-level `FundDeployer` can optionally implement migration hooks provided by `IMigrationHookHandler`, which give the release an opportunity to run arbitrary logic as the `Dispatcher` invokes those hooks at every step of the migration process.

As was the case described above with the `VaultLibBaseCore` , this abstracted notion of a `FundDeployer` - in which the `Dispatcher` only cares about its identity for access and for optional callbacks - is totally unrestrictive to the shape of the release-level protocol.

---

Release-level contracts, then, are mostly arbitrary from the standpoint of these persistent contracts, offering maximum flexibility for future iterations and changes.

# Release Architecture

### Core

**FundDeployer** The `FundDeployer` is the gateway for a user to create a new fund and signal the migration of a fund from an old release to the current release.

As described in the previous page, this is the contract that the `Dispatcher` considers as the `currentFundDeployer`, thus allowing it to deploy and migrate `VaultProxy` instances.

The `FundDeployer` deploys configuration contracts ( `ComptrollerProxy` ) per-fund that are then attached to `VaultProxy` instances (more in the next section).

The `FundDeployer` is also used as a release-wide reference point for a couple types of values.

The first is the `releaseStatus` . When set to `Paused` , any functions that write to `VaultProxy` storage are prohibited, other than redeeming shares. This is a safety mechanism in the case that a critical bug is discovered in one of the protocol contracts.

The second storage value of `FundDeployer` that is referred to release-wide is a registry of allowed "vault calls" , which is referred to by the `ComptrollerLib`. This allows for arbitrary calls to be made from the `VaultProxy` as `msg.sender` , which will be used in calls such as delegating the `SynthetixAdapter` as an approved trader of Synths directly from the `VaultProxy` .

There is 1 shared `FundDeployer` for the release.

**ComptrollerProxy** One `ComptrollerProxy` is deployed per-fund, and it is the canonical contract for interacting with a fund in this release. It stores core release-level configuration and is attached to a `VaultProxy` via the latter's `accessor` role described on the previous page.

All state-changing calls to the `VaultProxy` related to the fund's holdings and shares must thus pass through the `ComptrollerProxy`, making it a critically important bottleneck of access control.

The storage and logic of the `ComptrollerProxy` are defined by the `ComptrollerLib` and its associated libraries. Though it inherits the same upgradable `Proxy` as the `VaultProxy`, there is no way to call for an upgrade in this release.

**VaultLib** As described on the previous page, the `VaultLib` contract contains the storage layout, event signatures, and logic for `VaultProxy` instances that are attached to this release.

There is 1 shared `VaultLib` for the release.

### Extensions

Extensions extend the logic of the core contracts by adding additional kinds of functionality.

They are semi-trusted, in that they are selectively granted access to state-changing calls to `VaultProxy` instances.

In order to make such a state-changing call, two conditions must be met:

1. The Extension function must have been called by a `ComptrollerProxy` via a function with the `allowsPermissionedVaultAction` modifier, which opens the calling `ComptrollerProxy` to `VaultProxy` state changes.
2. The state-changing call must pass back through the `ComptrollerProxy`, and is delegated to the `PermissionedVaultActionLib` to determine whether the calling Extension is allowed to perform such an action.

This paradigm assures that an Extension can only perform a state-changing action to a `VaultProxy` if it was called by that `VaultProxy`'s corresponding `ComptrollerProxy` and if the Extension is permitted to make such a change at all.

Though this might seem like overkill for the current release, where extensions are all trusted and audited, it reduces the auditing surface area (e.g,. `PolicyManager` has no permitted actions) and opens the door for a subsequent release to allow arbitrary Extensions.

In this release, there are three Extensions. All funds share one contract per Extension.

**IntegrationManager** The `IntegrationManager` allows exchanging a fund's assets for other assets via "adapters" to DeFi protocols (e.g., Uniswap, Kyber, Compound, Chai).

It treats these adapter plugins in an almost untrusted manner (it does rely on adapters to report the expected assets to spend and receive based on user input), validating asset values spent and received against expected values, and also providing the opportunity for policies to implement hooks that are run pre- and post-asset exchange.

**PolicyManager** The `PolicyManager` allows state validation via "policies" that implement hooks invoked while buying shares and making an exchange in the `IntegrationManager`

There is no trust involved in policies, as the `PolicyManager` has no access to state-changing vault actions.

**FeeManager** The `FeeManager` allows for "fees" to dictate the minting, burning, or transferal of fund shares, according to their internal logics.

Like the `PolicyManager`, the `FeeManager` invokes hooks at different points in core logic, namely while buying shares, redeeming shares, and upon a specific function to invoke a "continuous" hook (e.g., for a `ManagementFee` that grows every block).

### Plugins

Each of the Extensions above make use of plugins. The `IntegrationManager` uses "adapters", the `PolicyManager` uses "policies", and the `FeeManager` uses "fees".

Allowed plugins are all defined on registries in their respective Extensions.

As with Extensions, the plan for subsequent releases is to open up these plugins for third party development.

### Infrastructure

In addition to "core" and "extension" release-level contracts, there are also entirely decoupled "infrastructure" contracts that can theoretically be recycled between releases. Currently, this category only contains contracts that are related to asset prices and values, but it can also contain contracts such as a forthcoming release that will implement protocol fees.

**ValueInterpreter** The `ValueInterpreter` is the single point of aggregation of various price feeds that are used to calculate the value of one or many input asset amounts in terms of an output asset.

There are two categories of assets in this release:

- "primitives" - assets for which we have direct rates with which to convert one primitive to any other (e.g., WETH, MLN, etc)
- "derivatives" - assets for which we only have rates in terms of underlying assets (e.g., Chai, Compound cTokens, Uniswap pool tokens, etc)

The `ValueInterpreter` determines whether an asset is a primitive or derivative, and executes logic to use corresponding price feeds to determine the value in the output asset.

There is only one supported price feed for each category in this release, so both are hardcoded as `immutable` variables.

**ChainlinkPriceFeed (IPrimitivePriceFeed)** The `ChainlinkPriceFeed` provides all conversions between primitives. This feed registers assets with their Chainlink aggregators, thus defining the primitive asset universe for the release.

**AggregatedDerivativePriceFeed** The `AggregatedDerivativePriceFeed` serves as a central registry that maps derivatives to their corresponding price feeds, and fetches rates from them.

There are several individual price feeds that provide the actual rates of derivatives to their underlying assets. Each inherits `IDerivativePriceFeed` to provide a standard interface for the `AggregatedDerivativePriceFeed` to register derivative mappings and grab rates. e.g., `CompoundPriceFeed`, `ChaiPriceFeed`

### Interfaces

All interfaces to external contracts are contained in the `release/interfaces/` directory.

Interfaces for internal contracts (e.g., `IFundDeployer` ) are kept beside the contracts to which they refer. These are narrow interfaces that only contain the functions required by other non-plugin, release-level contracts (i.e., those in the "core" and "extensions" sections above). The idea was to have a handy visual reference to the intra-release surface area of interactions between contracts.

## End Users

There are two primary end users of the Enzyme Protocol: fund owners and investors.

The Enzyme Protocol is designed to have protections in place for both of these parties without being overly restrictive and rigid.

### Fund owners

Each fund has an owner, stored on the `VaultProxy`, and thus persisting between migrations to new releases.

A fund owner configures the rules of their fund: fees and policies, the denomination asset by which share price and performance are measured, the time-lock between shares actions (buying or redeeming shares) for a given user, etc.

The fund owner exchanges assets in their fund for other assets via integration "adapters", e.g., `KyberAdapter` or `CompoundAdapter` . This is the primary way in which a fund owner can accrue value for their fund.

A fund owner can also assign a couple other roles to help manage their fund:

- a `migrator` who can migrate the fund to the current release
- accounts that are allowed to make asset exchanges via the `IntegrationManager` on behalf of the fund (e.g., a DAO fund owner can approve accounts to make exchanges on behalf of its funds)

### Investors

A fund can theoretically have unlimited investors, who get exposure to a fund's performance by buying and redeeming fund shares.

Shares are currently non-transferrable between investors (though they can be transferred within the protocol to pay fees).

#### Relationship between Fund owners & Investors

Fund owners can opt in to use one or more policies and other configurable measures to protect their investors where necessary. It is assumed that investors will pay special attention to review the protective policies & protections in a particular fund before investing. See "Known risks & mitigations" section.

## Administrators

There are two primary parties who administrate privileged functions in Enzyme Protocol: Avantgarde Core (deployer) and the Enzyme Council (admin).

#### Avantgarde Core (deployer)

As the lead developer of the protocol, Avantgarde Core deploys all contracts, and configures all of them prior to taking a release live. Once the release is taken live, full access control of protected functions is handed over to the Enzyme Council.

This procedure will be detailed in a later section.

#### Enzyme Council DAO (admin)

The Enzyme Council DAO is made up of two sub-committees; the Enzyme Technical Council (ETC) and the Enzyme Exposed Businesses (EEB). The ETC is a confederation of technically skilled appointed parties who together have sole access (via Aragon) to voting on all protected, protocol-wide functions once a release is taken live.

The Enzyme Council is a fully trusted entity, which is core to the security assumptions of the protocol.

There are currently 9 members of the ETC.

## Access Control Handoff

#### Dispatcher ownership

The owner of the `Dispatcher` is the canonical global admin for the protocol, persisting across releases (how the release implements that authority is up to the release).

When the `Dispatcher` is deployed, the deployer (Avantgarde Core) is its initial `owner`.

After configuring the protocol for the first release (both persistent and release-level contracts), Avantgarde Core initiates the first nomination-claim procedure to transfer `Dispatcher` ownership to the ETC:

1. Current `owner` calls `Dispatcher.setNominatedOwner()` with the ETC multi-sig address
2. The ETC votes to call `Dispatcher.claimOwnership()`
3. `owner` is set to the ETC multi-sig

This process can be repeated in the case that the ETC needs to change multi-sigs, or in the more extreme case of a governance model change.

#### FundDeployer ownership

For this release, the owner of the `FundDeployer` is taken to be the admin of release-level protocol contracts.

The owner of `FundDeployer` is set dynamically:

- when its `relaseStatus` is `PreLaunch`, the owner is the contract's deployer, i.e., Avantgarde Core
- when the `releaseStatus` is set to `Live` (which is only changeable by the `Dispatcher` owner), the contract then defers ownership to the owner of `Dispatcher`

This essentially creates a similar model at the release-level of a handoff between Avantgarde Core and the Enzyme Council.

#### Extensions and plugins ownership

Extensions (`FeeManager`, `PolicyManager`, `IntegrationManager`) and plugins (fees, policies, integration adapters) that require ownership for access control defer ownership to the current `FundDeployer` owner. This is accomplished by inheriting a `FundDeployerOwnerMixin` contract.

Thus when the owner of the `FundDeployer` becomes the ETC, so does the owner of all contracts that implement this mixin.

#### Infrastructure ownership

Price feeds, the ValueInterpreter, and any other future infrastructure that can potentially persist between multiple releases inherit a `DispatcherOwnerMixin`, if ownership is necessary. This contract operates exactly like `FundDeployerOwnerMixin`, but defers ownership to the `Dispatcher` owner.

This is necessary to leave infrastructure completely decoupled from any specific release.

---

These patterns of handing-off access control gives maximum flexibility for deployment and configuration, while assuring that the ETC will end up with complete admin privileges once the protocol is taken liven.

## ComptrollerProxy Creation

This is a common step for both fund creation and fund migration. It can be triggered by calling either `FundDeployer.createNewFund()` or `FundDeployer.createMigratedFundConfig()` , respectively.

Both of these functions create all release-level fund configuration at their outsets. This ordered steps are:

1. The `FundDeployer` deploys a new `ComptrollerProxy` instance, which sets the caller-provided core config via `ComptrollerLib.init()`
2. If the caller has provided configuration for an Extension (fees or policies), it then calls `ComptrollerLib.configureExtensions()` with the caller-provided config.

## Fund Creation

In order to create a new fund, CallerA (any account) can call `FundDeployer.createNewFund()` . The steps taken by this function are:

1. A `ComptrollerProxy` and full release-level fund configuration are created via the pipeline described in "ComptrollerProxy Creation".
2. `FundDeployer` calls to the `Dispatcher` to deploy a new `VaultProxy` with the CallerA-provided `fundOwner` and `fundName` (note that `fundOwner` does not need to be CallerA), along with the release's `VaultLib` and the newly-created `ComptrollerProxy` that will become the `VaultProxy` 's `accessor`.
3. The `FundDeployer` calls `ComptrollerProxy.activate()` to set the newly-created `VaultProxy` on the `ComptrollerProxy` and to give extensions a final chance to update state before the the fund can start taking investments.

## Fund Migration

### Philosophy

The most important, uncompromisable tenet of our migration pattern:

**One bad release must never be able to render a `VaultProxy` unmigratable to any future release.**

This is achieved through a series of mitigations, chief among them:

- Calls to migrate always come from the inbound `FundDeployer` , rather than vice-versa.
- Hooks that call down to the inbound and outbound `FundDeployer` instances must be able to be bypassed in the case of failure.

#### Step-by-step

In order to migrate a fund from a previous release to the current release:

1. CallerA calls `FundDeployer.createMigratedFundConfig()`. This deploys a `ComptrollerProxy` and sets all release-level fund configuration, as described in "ComptrollerProxy Creation".
2. CallerA calls `FundDeployer.signalMigration()` with the addresses of the `ComptrollerProxy` and `VaultProxy` that should be joined.
3. The `FundDeployer` validates that CallerA was the creator of the `ComptrollerProxy` and is a valid migrator for the `VaultProxy` .
4. The `FundDeployer` calls up to `Dispatcher.signalMigration()` , which stores a `MigrationRequest` with the passed values along with the `executableTimestamp` (the timestamp at which the migration will be allowed to be executed, based on the `migrationTimelock` value set on `Dispatcher` at the time migration is signaled).
5. After the current block's timestamp is greater than or equal to the `executableTimestamp` , CallerA can call `FundDeployer.executeMigration()`, which calls the mirroring function on the `Dispatcher` .
6. The `Dispatcher` validates whether the `migrationTimelock` has elapsed for the `MigrationRequest` , and whether the calling `FundDeployer` is still the `currentFundDeployer` (migrations to stale releases are not allowed).
7. The `Dispatcher` calls `setVaultLib()` and `setAccessor()` in order on the `VaultProxy`, updating the target of the proxy and the `accessor` role.
8. The `FundDeployer` calls `ComptrollerProxy.activate()` to set the migrated `VaultProxy` on the `ComptrollerProxy` and to give extensions a final chance to update state before the the fund can start taking investments.

#### Migration Timelock

As stated in the pattern above, there is a `migrationTimelock`, which defines the minimum time that must elapse between signaling and executing a migration. This gives investors the opportunity to opt-out of a fund if they do not agree to the upgrade, or to the new fund configuration.

#### Hooks and "emergency" functions

The `Dispatcher` invokes two types of hooks that call down to the outbound and inbound `FundDeployer` instances during the migration process, giving them the chance to execute arbitrary code at the release-level:

- `invokeMigrationOutHook` is called on the outbound `FundDeployer` instance before and after each action in the migration pipeline: signal, migrate, and cancel (only post-cancellation)
- `invokeMigrationInCancelHook` is called on the inbound `FundDeployer` instance post-cancellation. This is necessary because while the inbound `FundDeployer` is the caller in all other cases, if an approved migrator calls `FundDeployer.cancelMigration()` directly, the inbound `FundDeployer` should be given the opportunity to react.

These hooks are not guaranteed to succeed, but - as stated above - they must never block a migration.

This is why each migration function has a `bool _bypassFailure` param on the `Dispatcher`, which is set to `true` via `xxxEmergency` versions of each function on the `FundDeployer` , e.g., `signalMigrationEmergency()`

## Migration in this Release

It might seem odd that this release contains functions for both inbound and outbound migration, even though this is the first release of Enzyme v2.

Indeed, there can be no inbound migrations to the first deployment of this release, as legacy v1 funds are un-migratable.

The reasons for including inbound migration functions here are:

- to put inbound migration logic into practice for architecting and testing
- to get inbound migration logic audited, so that in the case of critical bug discovery, we can quickly iterate to a new release deployment without writing inbound migration logic from scratch

If it's the recommendation of the audit, we can remove (or comment out) all release-level inbound migration logic for the production deployment of this release.

## User Roles

For descriptions of user roles, see the "USERS" section.

**Trust Levels**

**Investors** Untrusted

**Enzyme Council** Fully trusted

**Fund Owners** Untrusted by default, but with on- and off-chain mitigations in place.

The Enzyme Protocol aims to mitigate against malicious fund owners within reason, while not being overly restrictive. This happens partially in the protocol, partially in the Enzyme Council's registry of plugins, and partially via off-chain trust mechanisms. For example,

- (protocol) the fund owner cannot arbitrarily act upon the VaultProxy (i.e., the fund's holdings or shares issuance)
- (protocol) the fund owner can only transact with a fund's assets via officially-registered adapters in the `IntegrationManager`
- (Council) only adapters that are not easily game-able are made available to all funds (e.g., only AMMs, rather than open order books; also, OTC trading is only allowed via Council-approved order makers)
- (Council) policies governed by the `PolicyManager` are continuously developed to programatically increase trust in a fund owner, e.g., policies that guarantee daily redemption windows, limit the permitted use specific of adapters, or limit asset composition in the fund's holdings
- (off-chain) the verification of fund owner identities

Investors should view fund managers as an untrusted foundation, upon which layers of trust can be added via these mechanisms.

**Migrators (including the fund owner)** Untrusted

A migrator can create any new arbitrary fund configuration to which a fund will be migrated.

This untrusted nature is mitigated by the timelock (initially 48 hours) between signaling and executing fund migration.

The investor is expected to review the new configuration and decide whether or not to remain in the fund within the timelock period.

# Extensions and Plugins

Though all contracts for this release are fully audited and trusted, we would like to allow for third party development of plugins (sooner) and extensions (later) in upcoming releases.

For that reason, the release-level contracts are architected like a trust onion, with each layer outward being less trusted:

- core (e.g., Comptroller contracts, Vault contracts, fund deployment)
- extensions (e.g., `IntegrationManager`)
- plugins (e.g., integration adapters like `KyberAdapter`)

The core dictates the actions that each extension can take on `VaultProxy` state. E.g., in this release, no `VaultProxy` state-changing actions are allowed by the `PolicyManager`, whereas `IntegrationManager` and `FeeManager` are allowed limited sets of actions.

Additionally, any state-changing call to the `VaultProxy` must both originate from and return through the same `ComptrollerProxy` of the corresponding fund, guaranteeing that extensions can only act upon a particular `VaultProxy` when called by the fund itself.

Extensions can then define their own rules for local behavior, e.g., how its plugins work (plugins are defined and used by all current extensions, but are not essential to an extension). For the official extensions provided by the core protocol, plugins are designed NOT to have direct state-changing authority on core contracts or extensions, and all operate in their own context (i.e., they are not delegate-called).

#### E.g., `IntegrationManager` and an adapter

The `IntegrationManager` is an extension that defines rules for how "adapters" (its plugins) can use a fund's assets in interactions with external DeFi protocols. The `IntegrationManager` itself has access to state-changing functions on the `VaultProxy` related to spending and accounting for asset holdings. It provisions these holdings to its adapters as instructed by the user input in `callOnIntegration()`. The rough pathway is as follows:

- permissioned user calls `IntegrationManager.__callOnIntegration()` via `ComptrollerProxy.callOnExtension()` with instructions to use AdapterA, SelectorB, and CalldataC
- `IntegrationManager` calls to AdapterA and asks for the assets and max amounts to be spent and the assets and min amounts to be received in the tx, along with how the assets to be spent should be provisioned to the adapter (e.g. approval or transferal)
- the `IntegrationManager` provisions the spend assets to the adapter as instructed
- the adapter uses the spend assets however it sees fit, generally in exchange for other assets, and then returns all unused spend asset balances and all incoming asset balances to the `VaultProxy`
- the `IntegrationManager` then runs validation on the amounts spent and received, and provides all final incoming and outgoing asset balances to the `PolicyManager` for further validation

One nuance that may become important (and changed) in an upcoming release is that the adapter (rather than the `IntegrationManager`) is responsible for parsing the assets to be spent and received from the user-input calldata, which has the potential for malicious abuse.

## Known Risks & Mitigations

`1.Flash attacks` **Description of attack**: It is possible to cheaply manipulate even the most liquid assets with the use of a flash loan. In the same block you can manipulate the price of a largely held fund asset (regardless of whether

liquid or not), buy/sell the fund advantageously to you, move the price back and return the loan in one transaction.

**Solution**: Mitigated by customizable time-lock between invest and redeem. The time lock is configurable by the manager and recommended to be **at least one second.**

`2.Arb attacks` **Description of attack**: Continuous invest/redeem at "stale" NAVs (ie. front-running of the NAV). If done enough times this could drain the fund of its AUM at the expense of other investors

**Solution(s):** Clear communication and helpful warnings around this at the UI level, the use of the configurable time-lock on shares actions, and one or more optional recommended policies to help reduce/mitigate at a protocol level. Some policy options could include:

- The use of an investor whitelist (KYC) to spot and report 'bad actors'
- The use of a policy which imposes an investor whitelist if a certain investment takes GAV over a configurable amount deemed as too risky for arb-ing.
- The use of the "liquidity fee" policy (see below*)
- More options are being explored

`3.Exploiting funds with less liquid assets` **Description of attack:** When a fund holds a large position in a less liquid asset, it is potentially vulnerable to the manipulation of that underlying asset (which effectively can manipulate the NAV of the fund too).

**Solution:** Implementation of a high liquidity fee policy (see below*) for low liquidity funds.

`4. Draining a fund through trade manipulation` **Description of attack:** If a malicious manager does an OTC trade(s) with himself at the "wrong price" (eg. selling an asset to himself for zero) this could drain the fund of all its assets.
****

**Solution:** We have removed the use of open trading on 0x order books, and only make the `ZeroExV2Adapter`available for approved order makers (e.g., our OTC trading partners). All trading with anonymous parties can only happen at the best available price from AMMs.

---

***Liquidity fee policy -** This is a percentage fee on entrance into the fund which is indirectly paid to the entire fund (not the manager) by burning a portion of purchased shares. Such a mechanism aligns investors with the costs caused by purchase/redeem churn (having to liquidate and repurchase assets frequently) as a result of short-term investors in the fund. This fee is customizable by the

manager and can be quite high for a fund holding very illiquid assets. If you stay in the fund for a while you will on average make the fee back from other joiners/leavers who pay the fee to you. Essentially in aggregate the fee is paid by shorter term investors to longer term investors.
****

# Adapter Integratees

In order to exchange some of a fund's assets for other assets, an adapter generally integrates with one or more "integratees," i.e., endpoints at which to interact with a defi protocol such as Kyber, Uniswap, Compound, etc.

### ChaiAdapter

Integrates with Chai token.

Docs: https://github.com/dapphub/chai

Mainnet contracts:

- `Chai`: `0x06AF07097C9Eeb7fD685c692751D5C66dB49c215`

Functions and considerations:

- `lend()` - None
- `redeem()` - None

### CompoundAdapter

Integrates with Compound Finance's cTokens. Each cToken is its own integratee.

Docs: https://compound.finance/docs

Mainnet contracts:

- `cBAT` : `0x6c8c6b02e7b2be14d4fa6022dfd6d75921d90e4e`
- `cCOMP`: `0x70e36f6bf80a52b3b46b3af8e106cc0ed743e8e4`
- `cDAI`: `0x5d3a536E4D6DbD6114cc1Ead35777bAB948E3643`
- `cETH`: `0x4ddc2d193948926d02f9b1fe9e1daa0718270ed5`
- `cREP`: `0x158079ee67fce2f58472a96584a73c7ab9ac95c1`
- `cUNI`: `0x35A18000230DA775CAc24873d00Ff85BccdeD550`
- `cUSDC`: `0x39aa39c021dfbae8fac545936693ac917d5e7563`
- `cZRX`: `0xb3319f5d18bc0d84dd1b4825dcde5d5f7266d407`

Functions and considerations:

- `lend()` - fund receives `cToken` , which triggers the `VaultProxy` to start accumulating `COMP` based on the amount lent. `COMP` is claimable natively on Compound on behalf of the fund (by any user)
- `redeem()` - None

### KyberAdapter

Integrates with Kyber Network via the `KyberNetworkProxy`.

Docs: https://developer.kyber.network/docs/Start/

Mainnet contracts:

- `KyberNetworkProxy` : `0x9AAb3f75489902f3a48495025729a0AF77d4b11e`

Functions and considerations:

- `takeOrder()` - None

### ParaswapAdapter

Integrates with Paraswap via the `AugustusSwapper`. Incorporates asset approvals via the `TokenTransferProxy`.

Docs: https://etherscan.io/address/0x9509665d015bfe3c77aa5ad6ca20c8afa1d98989#code

Mainnet contracts:

- `AugustusSwapper`: `0x9509665d015Bfe3C77AA5ad6Ca20C8Afa1d98989`
- `TokenTransferProxy`: `0x0A87c89B5007ff406Ab5280aBdD80fC495ec238e`

Functions and considerations:

- `takeOrder()` - None

### SynthetixAdapter

Integrates with Synthetix via `SNX`.

Docs: https://docs.synthetix.io/

Mainnet contracts:

- `SNX`: `0xC011a73ee8576Fb46F5E1c5751cA3B9Fe0af2a6F`

Functions and considerations:

- `takeOrder()` - Synthetix gives the incoming asset of the trade a provisional "best guess" balance until it reaches finality, which is after the Synthetix-defined "waiting period" (currently 3 minutes) has expired. The final balance can be +/- the provisional balance.

### TrackedAssetsAdapter

Does not integrate with any external contracts.

It only serves to add tracked assets to a fund in a manner that subjects the additional to policy management.

Functions and considerations:

`addTrackedAssets()` - None

#### UniswapV2Adapter

Integrates with UniswapV2 for trading and for liquidity provision and redemption.

Docs: https://uniswap.org/docs/v2/

Mainnet contracts:

- `UniswapV2Factory`: `0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f`
- `UniswapV2Router2`: `0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D`

Functions and considerations:

- `takeOrder()` - None
- `lend()` - None
- `redeem()` - None

#### ZeroExV2Adapter

Integrates with the 0x Protocol.

Docs: https://github.com/0xProject/0x-protocol-specification/blob/master/v2/v2-specification.md

Mainnet contracts:

- `Exchange`: `0x080bf510fcbf18b91105470639e9561022937712`
- `ERC20Proxy`: `0x95e6f48254609a6ee006f7d493c8e5fb97094cef`

Functions and considerations:

- `takeOrder()` - Left unchecked, the 0x Protocol allows filling any trade created by any user, irrespective of whether the resulting value is beneficial for the fund. This adapter method is limited to makers approved by the Enzyme Council to mitigate this issue.

## Price Feed Sources

Just as adapters interact with external "integratees," so do price feeds interact with sources that provide them with the data they need to provide rates.

#### ChainlinkPriceFeed

Each rate pair (we use those quoted in either USD or ETH) is provided by a Chainlink "aggregator," and we interact with the proxy contracts for those aggregators.

Docs: https://docs.chain.link/docs/using-chainlink-reference-contracts

Mainnet contracts: each aggregator

Considerations:

We do not check timestamps on every price lookup, rather we provide a function to allow quickly removing an aggregator that is considered to be stale. If an aggregator is removed, the corresponding primitive will cease to produce a price from the feed and will revert, causing any function that relies on that price to fail until it is re-instituted. This is the desired behavior.

### ChaiPriceFeed

This price feed mimics how `Chai` determines its rate in its codebase, by interacting with Maker DAO's DSR `Pot` contract.

Docs: https://github.com/dapphub/chai

Mainnet contracts:

`Pot` - `0x197E90f9FAD81970bA7976f33CbD77088E5D7cf7`

Considerations:

We do not call `Pot.drip()` before calculating price as in `Chai`, in order to save gas (`drip()` updates storage). The CHAI-DAI rate changes negligibly for even long periods of time.

### CompoundPriceFeed

Queries each Compound Token (cToken) directly for its rate.

Docs: https://compound.finance/docs

Mainnet contracts: each cToken

Considerations:

We query the cached rate instead of the live rate for gas savings. Like `CHAI`, cTokens rates change negligibly for long periods of time.

### SynthetixPriceFeed

Queries the Synthetix `Exchanger` contract to get the rate for a Synth in terms of `sUSD` .

Docs: https://docs.synthetix.io/

Mainnet contracts:

- `AddressResolver` - `0x61166014E3f04E40C953fe4EAb9D9E40863C83AE`

Considerations: Live prices

#### UniswapV2PoolPriceFeed

Uses a special pool manipulation-resistant formula that takes into consideration the current underlying asset balances and pool token balance in the given Uniswap pool (`UniswapV2Pair`) along with a trusted rate between the two underlying tokens.

Docs: https://uniswap.org/docs/v2/

Mainnet contracts:

- `UniswapV2Factory`: `0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f`
- instances of `UniswapV2Pair`

See also the sample implementation we based this on: https://github.com/Uniswap/uniswap-v2-periphery/blob/267ba44471f3357071a2fe2573fe4da42d5ad969/contracts/libraries/UniswapV2LiquidityMathLibrary.sol

Considerations: Live prices

#### WDGLDPriceFeed

This price feed uses a known formula for pricing DGLD based on the current rate of XAU. It uses Chainlink aggregators directly.

Docs: https://dgld.ch/

Mainnet Contracts:

- XAU/USD Aggregator: `0x214eD9Da11D2fbe465a6fc601a91E62EbEc1a0D6`
- ETH/USD Aggregator `0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419`

Considerations: Live prices

## Management Fee

The current management fee implementation is making some approximations, which are not necessary. Instead, we can calculate the correct number based on the formulas below.

### Some definitions

Management fee rate (annual, in percent):`x`

Effective management fee rate (annual, in percent, after dilution): `k`

Since management fee is not paid out (as a percentage of assets), but is allocated as newly minted shares in the fund, we need to use the effective management fee rate. This ensures that the manager receives the correct ratio of shares.

The two fee rates are related as follows:

$$x = \frac{k}{1+k}$$

or, alternatively

$$k = \frac{x}{1-x}$$

## Continuous compounding

Management fee accrual happens at irregular and unknown intervals, so we have to resort to continuous compounding. The continuous management fee rate `z` is related to the annual effective management fee rate `k` as follows:

$$e^z = 1 + k$$

or, alternatively

$$z = ln(1+k)$$

Substituting for the effective management fee rate `k` yields the relation between the continuous management fee rate and the annual management fee rate:

$$e^z = \frac{1}{1-x}$$

or, alternatively

$$z = -ln(1-x)$$

## Management fee allocation

Whenever management fee is due after a time period `t` (expressed as a fraction of a year), the number of shares changes as follows

$$S' = e^{z \cdot t} S$$

`S` is the total supply of shares before the allocation of the management fee shares, and `S'` is the total supply of shares after the allocation of the management fee shares.

The share allocation to the manager is `S_{manager} = S'-S`, and it is calculated as follows:

$$S_{manager} = \left( \frac{1}{(1-x)^t} - 1 \right) S$$

or

$$S_{manager} = \left( (1+k)^t - 1 \right) S$$

Using `t = \Delta t / N`, we can rewrite this as

$$S_{manager} = (f^{\Delta t} - 1)S$$

where

$$f = (1+k)^{1/N}$$

`f` is calculated off-chain when configuring the fee, and it is stored on-chain as `scaledPerSecondRate` . The on-chain computation is then

```
sharesDue = (rpow(scaledPerSecondRate, numberOfSeconds, 10*27) -
10**27) * totalSupply / 10**27
```

# Performance Fee

## Principles

- Performance fee can be calculated easily and correctly as long as all investors invest and redeem at the same time (i.e. as long as the number of shares in the fund remains constant over time).
- If investors enter and exit at different points in time, then that can lead to unequal treatment of investors and also of the fund manager. This is due to the fact that the number of shares is not constant over time, and we either have to assume certain values (e.g. the current model assumes that the number of shares throughout the period is equal to the number of shares at the end of the period) or we have to keep track of the exact entry and exit points of each share (which requires a lot of on-chain storage).
- Performance fee crystallises only after a certain period of time (e.g. after one year or one quarter). This crystallisation period acts as a buffer for short-term fluctuations, and incentivises the fund manager to optimise for long-term performance and not for short-term fluctuations.
- Performance fee is paid out by creating shares for the manager, conserving the number of shares that investors hold.

## Implementation

- We divide the fund lifetime into periods with constant number of shares, so called *constant share periods*. We calculate the performance fee at the end of each *constant share period*. A constant share period ends, whenever an investor buys shares or whenever an investor redeems shares.
- Performance fees calculated at the end of a *constant share period* are allocated into an account, the *AggregateValueDue* account (which is a liability on the fund's balance sheet).
- The *AggregateValueDue* account can increase or decrease, depending on the performance and the number of shares during a *constant share period*. It cannot become negative.
- In addition to accounting for *AggregateValueDue*, shares corresponding to that amount are minted (or burned, depending on the performance), and they are held by the VaultProxy itself. Those shares are called *sharesOutstanding*. Those *sharesOutstanding* will be transferred to the manager at the end of the crystallization period (see below)
- Performance Fee can be paid out to the manager at the end of the crystallisation period (generally a year, or a quarter). The manager has to initiate the transaction. If the *AggregateValueDue* account has a non-zero balance, the corresponding number of *sharesOutstanding* is paid out and HWM is set to the new value

## Formulas

Assumptions:

- management fee is already paid out (shares have been minted and transferred to the manager)
- performance fee for previous periods is not paid out but accrued
- calculations below can be carried out in the order written down
- formulas are written in LaTeX notation

Performance fee percentage:

$$x\%$$

Period number:

$$i$$

Gross asset value (value of assets in fund):

$$GAV$$

Number of shares (before investment or redemption - EXCL. phantom shares):

$$TS_i$$

High watermark:

$$h_i$$

**Calculate performance fee due** Previous gross share price (read from storage, initially 0):

$$g_{i-1}$$

Current gross share price (i.e. net of management fees, but gross of performance fees due for period)

$$g_i = GAV_i / TS_i$$

Wealth above HWM created during period:

$$W_i = (max(h_i, g_i) - max(h_i, g_{i-1})) \cdot TS_i$$

Performance fee during period:

$$F_i = W_i \cdot x\%$$

---

*AggregateValueDue* carried over from previous period (read from storage):

$$AF_i = max(0, AF_{i-1} + F_i)$$

*AggregatedValueDue* after period (write to storage):

$$AF_i = max(0, AF_{i-1} + F_i)$$

NAV after accrual of performance fee:

$$NAV_i = GAV_i - AF_i$$

Net share price after accrual of performance fee:

$$s_i = \frac{NAV_i}{TS_i}$$

**Issue sharesOutstanding** Calculate *sharesOutstanding* (equivalent of *AggregatedValueDue*):

$$O_i = \frac{AF_i \cdot TS_i}{GAV_i - AF_i}$$

Load previously minted *sharesOutstanding*:

$$O_{i-1}$$

New shares to be minted (burn if negative)

$$O_i - O_{i-1}$$

Net share price (different formula, same value as above):

$$s_i = \frac{GAV_i}{TS_i + O_i}$$

**Investment** Performance fee is accrued (see above formulas)

New investor buys `n` shares at net share price `s_i` for a total value of

$$V_i = n \cdot s_i$$

New total supply:

$$TS_i' = TS_i + n$$

New GAV:

$$GAV_i' = GAV_i + V_i$$

**or: Redemption** Performance fee is accrued (see above formulas)

Investor redeems `n` shares at net share price `s_i` for a total value of

$$V_i = n \cdot s_i$$

If performance fee is due (i.e. *AggregatedValueDue* is larger than zero), the redeemers portion of the *sharesOutstanding* is transferred to the manager. New number of phantom shares is

$$O_i' = (1 - x\%) \cdot O_i$$

*AggregatedValueDue* in storage is reduced by the redeemers portion:

$$AF_i' = (1 - x\%) \cdot AF_i$$

New total supply:

$$TS_i' = TS_i + n + x\% \cdot \hat{s}_i$$

New GAV:

$$GAV_i' = GAV_i - V_i$$

**After investment/redemption** Calculate gross share price (excl. *sharesOutstanding*) (write to storage):

$$g_i = GAV_i'/TS_i'$$

**End of Crystallisation Period / Performance Fee Payout** Performance fee is accrued (see above formulas)

If the number of *sharesOutstanding* is greater than zero:

1. All *sharesOutstanding* are transferred to the manager.
2. *AggregateValueDue* amount is set to zero.
3. HWM `h_i` is set to new value `h_i = g_i = s_i` (after payout, gross share price and net share price are equal, and HWM is set to that price)

If accrued performance fee is zero:

1. Do nothing

**Links** esma_34-39-968_final_report_guidelines_on_performance_fees.pdf

UCITS%20Performance%20Fees%20–%20ESMA's%20call%20for%20harmonisation.pdf

# Release-Level Pause

In order to protect against value loss after critical bug discovery, this release allows the `Dispatcher` owner (i.e., the ETC) to institute a release-wide pause by setting `FundDeployer.releaseStatus` to `Paused` .

While in a `Paused` state, there are no calls allowed to any functions that change `VaultProxy` state (i.e., to protect value), except for shares redemption functions.

The idea is that funds would then migrate to a newly deployed release, potentially with a temporarily shortened `migrationTimelock`

The imagined pipeline of bug discovery would be:

1. The bug finder reports said bug to the Enzyme Council
2. If the bug is deemed critical, the ETC votes to `setReleaseStatus` for that release to `Paused`
3. Avantgarde Finance immediately fixes the bug
4. The bug fix is audited by either an auditing partner or a combination of auditors on the ETC
5. A new release is deployed, and fund managers are instructed to migrate to the new release

Recognizing that it may take some time between pause and deployment of a new release and that not all bugs may be considered critical for all funds, we have provided a per-fund option to `overridePause` . The fund owner can set this to `true`, which will allow all fund operations to proceed as normal.

The fund owner must thus make the ultimate determination on the safety of unpausing operations for their fund.

## Continuous Shares Redeemability

One of the major goals of the Enzyme Protocol is to make shares continuously redeemable. i.e., an investor should be able to redeem any amount of shares at any point in time and receive their proportionate slice of fund assets instantly, without needing to worry about the limited availability of liquid assets.

The core contracts facilitate continuous redeemability, and indeed it is important that the `redeemShares()` and `redeemSharesDetailed()` functions are kept relatively simple with few external calls, to reduce the chance of a redemption-blocking error.

It should also be noted, however, that there is no core guarantee on being able to redeem shares for a full compliment of assets at any given time, as this is contingent on the fund manager's use of configurations and asset universe:

- e.g., a fund manager can specify a `sharesActionTimelock` , which requires that a specified amount of time must elapse between buying shares and redeeming shares.
- e.g., a fund manager can trade assets that do not have instant finality. Synths are one example of this, which do not achieve a final balance after a few minutes, and are not transferable until they do so. In fact, a malicious fund manager could exploit this non-redeemable period to continuously trade Synths and block redemption indefinitely, but a properly-configured policy can mitigate this risk. For example, when configured properly, the `GuaranteedRedemption` guarantees a redemption window daily.

It is thus important for the investor to consider the fund's configuration when determining if a fund manager has provided adequate guarantees for consistent redeemability.

## Shares Requestors

A "shares requestor" is a contract that receives requests to buy shares in a fund, custodies the asset necessary to buy shares, and then features some mechanism for executing the requests.

A fund would generally use a shares requestor in tandem with the `BuySharesCallerWhitelist` policy, so that only the shares requestor contract has access to `ComptrollerProxy.buyShares()` .

#### AuthUserExecutedSharesRequestor

The first shares requestor included in this release.

It is a shares requestor in which submitted requests are manually executed by authenticated users (either the fund owner or fund owner-appointees).

Each fund gets their own proxy, which will generally be deployed via the provided factory.

**Setup** The pipeline to setup the shares requestor is:

- The fund owner deploys a `AuthUserExecutedSharesRequestorProxy` via the `AuthUserExecutedSharesRequestorFactory`
- The fund owner adds the `BuySharesCallerWhitelist` policy with the newly deployed`AuthUserExecutedSharesRequestorProxy` as the only caller
- The fund owner adds authorized users who can execute requests by calling `addRequestExecutors()` on their proxy

**Creating, executing, canceling requests** The pipeline for requests is:

- An investor creates a request to buy shares by calling `createRequest()` with the amount of the fund's denomination asset they wish to spend and the minimum amount of shares they will accept for that amount. The shares requestor proxy custodies the investment asset.
- A fund owner or auth user calls `executeRequests()` with the investor addresses of the requests they wish to execute. This will purchase the shares in each request via `ComptrollerProxy.buyShares()`, and remove each stored request.
- An investor can cancel their request at any time and have their investment asset refunded.

- An investor can only have one request at any given time, and if they cancel a request, they must wait for a cooldown period before they are allowed to create another request.

**Fund migration notes** Since funds can migrate to new releases and since upon doing so the old ComptrollerProxy is no longer used, the shares requestor will also cease to function after a migration.

The expected behavior surrounding migration is:

- When a fund has a pending `MigrationRequest` in the `Dispatcher`, calls to `ComptrollerProxy.buyShares()` are prohibited.
- When a fund has migrated to a new release, calling `ComptrollerProxy.buyShares()` is also not possible for a multitude of reasons.
- Therefore, in both of these cases, the `exectueRequests()` function on the shares requestor will also fail.
- The shares requestor does not validate whether or not a fund is in either of these states when creating new requests, so it will continue to accept requests and allow cancellations, but will revert upon execution.

# Trading Synthetix Synths

There are a couple of key idiosyncratic behaviors to keep in mind for funds that trade and hold Synthetix synths that do not apply to the general asset universe, and there are certain mitigations for these behaviors that fund managers are encouraged to take to provide protective guarantees to their investors.

### Trading synths via the SynthetixAdapter temporarily blocks actions that transfer or burn the received synth

See the "Adapter Integratees" section for detail.

Note that these same restrictions do not apply for trading synths via 3rd party protocols such as Uniswap.

Recommended mitigation: use of the `GuaranteedRedemption` policy to provide a daily window during which investors will be able to redeem shares

### Liquidations can occur that will force the conversion of the synth into sUSD

A `PurgeableSynth` can be liquidated by the Synthetix team if certain conditions are met. This action burns the full synth balance and mints an equivalent amount of sUSD (minus fees) for the specified synth holders.

In such a case, the fund would receive sUSD and they would still have the liquidated Synth as a tracked asset, albeit with a 0 balance.

If the fund does not have sUSD as a tracked asset already, this opens the door to arbitrage the fund's (too low) share price, since the sUSD value will not be included in the fund's GAV.

Recommended mitigation: set sUSD as the fund's denomination asset (which guarantees it will always be tracked in the current release)

Alternative mitigation: always carry a nominal balance of sUSD (a less "trustless" solution as it relies on trusting the fund manager to not intentionally open the door for arbitrage)