Ithil

The Web3 Wizard

Ithil is a speculation primitive and core building layer that facilitates the creation of novel financial services.

V2.0.1 - February 24, 2023

Abstract

Ithil aims to become the base layer for decentralised financial services via a well-thought system of composable smart contracts, paired with liquidity vaults to issue debit or credit to.

Modular and easily upgradable, Ithil offers users and other protocols composable, audited lego blocks, enabling an entirely new range of financial opportunities to be created. External protocols can speed up their go-to-market by having an existing infrastructure and access to liquidity from day 1, investors can find novel solutions to speculate on and lenders can get diversifie exposure to the whole web3 space from a single platform.

Contents

1 Introduction

In the current Web3 space there are several opportunities, from DeFi high yields to NFTs and Real World Assets (RWA), with new ones coming up almost every day. Users are shown a wide landscape to interact with, depending on their risk appetite and personal ideas. However, all the subspaces are set apart one another, and end up being self-contained bubbles that don't interact with each other: DeFi, NFTs, RWA and metaverse are all separate from each other and the composability of one another is still uncharted territory.

While for the lenders, the typical parameters LPs look for are a good APY, the market exposure (such as by holding volatile tokens), the underlying protocol's reliability and security, the liquidity available or TVL, their personal preferences and several other factors difficult to predict or model. With Ithil they are presented a novel opportunity to get exposed to the high yields on the whole Web3 space in a combined way, reducing the overall risk as well as increasing the return rates on their deposits.

1.1 What is Ithil

At its core, Ithil is a protocol allowing *liquidity providers* to deposit their assets, and *users* to deploy such assets into external protocols within the Web3 world. Users need to protect the LPs' assets by placing some *collateral* and by paying *interests* on the deployed liquidity. In this way, LPs benefit of a very low risk and a solid return thanks to a high diversification and capital protection, while users are entitled to all earnings and services coming from the deployed assets.

1.2 What can be done on Ithil

With Ithil, anyone can

1. Become **liquidity provider** (LP) by depositing their assets to get a solid APY. A extensive choice of whitelisted tokens can be staked in Ithil, and the APY generated is in the same token as the provided one. In this way, Ithil offers an attractive staking opportunity, also for holders of volatile tokens.

2. Boost their investments by placing some collateral and then using the LP liquidity on one of Ithil's whitelisted protocols (Aave, Uniswap, Balancer, OpenSea etc...). Thanks to an internal system of undercollateralised loans (the **Internal Lending Engine**), the total capital deployed can be much higher than the collateral placed: by placing only 100 DAI worth of collateral, a user can deploy 1000 DAI worth of liquidity or more.

3. Be a **liquidator** by constantly checking open positions and liquidating thoe at loss in a fully decentralised way.

4. Become an **automator** by performing farming or other maintenance duties on the services which require them and getting a reward for that.

5. Join the **community** by holding and staking Ithil governance token, to take part in the protocol governance, earn part of the fees or trade.

1.3 Ithil's unique features

Many of Ithil features are scattered around the current web3 landscape and need users to combine multiple protocols to achieve the same result. Some of the differences are the following:

- **Modularity** allows Ithil to list or de-list virtually any service with a governance vote, hence continuously updating the services offered integrating the most recent protocols, always featuring what is best and trending, be it DeFi, Metaverse, Play to Earn or anything.
- **High capital efficiency** over-collateralisation has always been an essential aspect in DeFi loans, which reduces the possibility for users to leverage their available funds.[1] Thanks to a novel lending model, Ithil makes it possible to protect the loans with just the right amount of capital, thus allowing a completely new set of speculative financial services.
- **Opportunity to earn from virtually any token**: few protocols offer a single-sided APY on ideally any token, usually being restricted to stablecoins, high market cap tokens like WBTC and WETH or the protocol native token. Ithil lending vaults are instead token-agnostic: any ERC20 token can be whitelisted and lent, collecting fees in that same token.
- **Real-yield sustainable fee redistribution** thanks to a sustainable treasury management and a non-dilutive TVL boosting system.
- **Capital protection** for liquidity providers: an efficient liquidation model and proper risk tranching protect LPs' liquidity, thus assuring hedged earnings with limited risk exposure.
- **Solid tokenomics** thanks to Protocol Owned Liquidity (POL), call and put options on the token and fee accrual for buybacks, a positive feedback mechanism constantly bringing value to the protocol and to the token holders in a sustainable way. See Section **??** for details.

1.4 The Vision

Ithil main aim is to offer the broadest range of decentralised financial services for the web3 users and help get the most out of them.

By interconnecting the several protocols existing in the different niches of web3, Ithil can create innovative speculation opportunities (DeFi strategies), help users obtain what they want (on-chain mortgages) or simply have fun.

Through leverage, Ithil wants to mitigate the intrinsic advantage of wealthy individuals, allowing everyday people to fully embrace any opportunity web3 has to offer.

Building trust is a primary goal, especially in an ecosystem where scammers, Ponzi schemes, and ill-modelled speculative systems are unfortunately very popular. Ithil commits to giving real value to the community: every surplus earned by the protocol is algoritmically distributed to the token holders, thus increasing the value of the governance token and the financial power of its treasury in a simple and sustainable way.

[1] Or makes this process very costly, mainly through the so-called *folding strategies* on money markets like Aave.

2 Core concepts

This section summarises the inner workings of the protocol, which will be treated more in-depth in the following sections.

2.1 The Vaults

The **Vaults** are ERC4626 smart contracts that collect liquidity to be used in services required by the users. The liquidity contained in each Vault is made of a **whitelisted ERC20 token**, which can be in principle any token, from stablecoin to meme and rebasing tokens. The whitelisting process belongs to the governance.

Liquidity Providers (LPs) can freely **deposit** into the Vaults and **withdraw** from them at any time. The Vault contract collects the fees generated by the Services, which increase the share price of the Vault's ERC4626 token thus yielding an APY to the LPs.

The Vaults **lend** liquidity to the Debit Services (see 2.3.1) and **borrow** liquidity from the Credit Services (see 2.3.2) via internal uncollateralised loans. Just the Manager contract (see 2.2) can execute borrowing and lending on behalf of a specific Vault.

2.2 The Manager

The **Manager** is responsible for the general accounting and coordination between the Vaults and the Services. **Users** can access the Services through the Manager, which in turn transfers the Vaults' liquidity from or to the **whitelisted Services** contracts. The whitelisting process takes place via governance votes.

When a user accesses a Service, it is said that the user is **entering an Agreement**. Similarly, by **exiting an Agreement** the user restores the initial state of the Vaults plus fees.

In order to be able to access a Service, users may need to deposit a **collateral** to cover for potential losses incurred by LPs, whose liquidity is borrowed from or lent to the Service.

When liquidity is lent or borrowed, the Manager calculates the **interest rate** of the loan, based on parameters coming from the global state of Ithil core (free liquidity, displaced liquidity in loans, tranched reserves, collateral amount, token risk factors, etc...).

2.3 Services

By **Services** we define a smart contract that offers a specific financial resource by interacting with the Vaults' liquidity to perform a certain action. The services are broadly divided into **Debit Services (DS)** and **Credit Services (CS)** depending on respectively whether the service itself needs to borrow from or lend liquidity to the Vault.

When **entering an Agreement**, the user gets an **agreement NFT** which represents the user's entitlement to a certain asset held in the service and the liquidity taken from, or given to, the Vaults. *The Manager (see 2.2) maintains the ownership of such asset until the loan is repaid.* See 2.3.3 for further details about agreement NFTs.

When **exiting an agreement**, all the loans are extinguished, the due fees are paid to the Vault(s) and the agreement NFT is burned. The user gets any other assets remaining after the closure as profit.

See 6 for real examples of supported Services and Targets.

2.3.1 Debit Services

A **Debit Service** is a contract which triggers one or more loans *from a Vault* and deploys the obtained liquidity to an combination of other contracts called **Targets** (see 2.5). These targets are typically, but not necessarily, outside of Ithil: examples are OpenSea, Balancer, Aura, Curve, Aave, etc...

2.3.2 Credit Services

A **Credit Service** is a contract that lends liquidity *to a Vault* and deploys the obtained ERC4626 LP token to another contract called **Target** (see 2.5). These targets are typically, but not necessarily, within Ithil: examples are tranches, Insurance Reserves, $ITHIL Call and Put Options, etc...

2.3.3 Agreement NFT

Every time a user interacts with a Service, an **agreement NFT** representing the specific User-Service interaction is minted to the user wallet. The data contained depends on the particular service it refers to, but all services share some common parameters:

- The **owner**, i.e. the address of the user entering the service.
- The service **name**.
- The **loan** amounts.
- The type and number of assets hold in the service as **collateral**.
- The **interest rate** of the loan taken or given.
- The **timestamp** of when the agreement was subscribed.
- Some **extra parameters** depending on the particular service.

2.3.4 Quoter

Every service must have its own **quoter**, an on-chain function in charge of evaluating the capital health of each agreement. It is developed and part of the Service smart contract, so that each service has one and only one quoter.

By *evaluation* we mean the amount of Vault's tokens the user can obtain when exiting the agreement in that particular moment; this number is called the **value** of the agreement NFT. This includes the Target's payoff in that moment and the The value is used to compute the **liquidation score** of an agreement NFT: if the liquidation score becomes positive, any **liquidator** user (see **??**) can trigger a special function of the Manager to forcefully close the position and repay the loan before it becomes insolvent. An extra fee is then applied to the user margin in order to compensate the liquidator.

2.4 Fees

Agreements may or may not have **fees** attached. The fees are always deposited to a Vault and contribute to the appreciation of the ERC4626 token-shares representing the Vault deposits. Fees are paid when an agreement is *exited* but unlocked over time to prevent frontrunning attacks.

The fee structure of an agreement depend on the Service it refers to, in general we can identify three types of fees.

2.4.1 Interest rate

When liquidity is borrowed from a Vault, an **interest rate** is applied to the loan. The calculation of the interest rate is performed by a separate **interest rate library** and the result depends on the global state of Ithil. As a rule of thumb, the interest rate increases proportionally to the Vault liquidity usage and to the overall amount of liquidity already locked in that Service, while it decreases with the amount of user's collateral and with elapsed time from latest interaction (as in a Dutch auction scheme). The proportionality coefficients are decided by the governance and vary for each token and each Service, therefore the actual determination of the interest rate depends on a mixed algorithmic + governance-provided set of parameters.

2.4.2 Fixed fees

When a Service is used, a **fixed fee** might be charged to enter the Agreement. This fee may or may not depend on the amount of collateral posted or liquidity borrowed. The fixed fees are set by the governance.

2.4.3 Performance fees

Some Services (typically externally-created ones) may have a **performance fee**, where percentage of the accrued profits goes to the Vault.

2.5 Targets

By **Target** we mean any contract to which borrowed or lent liquidity is deployed to. Targets, which can be either external or internal to Ithil ecosystem, represent the way anyone can use Ithil to access external services with more liquidity than their original (leverage), or internal services to get extra benefits from lending their liquidity to Ithil. See 6 for real examples of supported Services and Targets.

2.6 Automators

Automators are external players which perform various tasks in order to optimise the outcome of each Service, and get rewards by doing this. Everybody can be an automator and there is no entry cost nor staking requirement, since Ithil considers the automators' actions as beneficial to the ecosystem.

As an example of automators we will talk about liquidators and harvesters.

2.6.1 Liquidators

A **liquidator** is anybody who makes an agreement with positive **liquidation score** to be forcefully exited. In this case, the liquidator becomes the owner of the agreement and gets the remaining funds as a reward. In case no liquidity is left, LPs in junior tranches are charged for the liquidation rewards due. In this way, liquidators ensure the loans are repaid in the fastest way as possible: even if a loss is inevitable, liquidation will be quickly performed and minimise it.

2.6.2 Harvesters

A **harvester** is anybody who triggers special functions to improve the performance of a Service and it is always rewarded using part of the extra value given to the Service.

This abstract definition is better explained with examples (see Section 6 for further details on these examples): the "Balancer + Aura" Service needs Aura harvesting to increase the Service overall APY via compounding and the harvester is rewarded with part of the extra liquidity given by Aura; the "Boosting" strategy needs continuous minting or burning of Vault's tokens to rebalance the Service inner value locked and the harvester is rewarded with part of such tokens.

The precise properties of harvesting greatly vary from Service to Service and it is directly in control of the developer.

3 Examples of Services

Examples of Services, with the respective Targets and the way they work, are the following.

3.1 Debit services

These services are characterised by liquidity that goes *from* the Vault *to* the particular service. In this case the Vault issues a loan to the Service by transferring the liquidity to the Service smart contract, which then locks any asset obtained. This loan is repaid when the agreement is exited.

- **Yearn**. Liquidity is taken from a Vault and deployed on Yearn; the resulting y-Tokens are then locked into the resulting agreement NFT. At the exit, the service's y-Tokens are redeemed on Yearn to repay the loan from the Vault.

- **Balancer + Aura**. Liquidity is taken from a Vault and deployed on a Balancer pool; the resulting BP-Tokens are then deployed on Aura and the resulting liquidity entitlement is given to the agreement NFT address. During the service's life, harvesters (see **??**) collect the Aura rewards and fill (via a reflaction) all open agreement NFT's. At the exit, the BP-Tokens are unstaked from Aura and redeemed on Balancer to repay the loan from the Vault.

- **Uniswap V3**. Liquidity is taken from one or two Vaults and deployed within a price range on Uniswap V3. The resulting UniV3 NFT is then locked into Ithil's agreement NFT. At the exit, the liquidity and generated fees are withdrawn from Uniswap NFT and the loan to the Vault is repaid.

- **OpenSea**. Liquidity is taken from a Vault to purchase an NFT from OpenSea, which is then locked into the agreement NFT. In order to keep the position open, the user must regularly inject liquidity into the Vault to partially repay the loan. At the exit, the loan is fully repaid and the original NFT is transferred to the user.

3.2 Credit services

These services are characterized by liquidity that goes *to* the Vault *from* the particular service. In this case the User provides the liquidity to the Service smart contract, which then transfers it to the Vault and gets the resulting Vault's token. The token is then used for the Service particular functionalities and it is redeemed to the Vault when the agreement is exited.

- **Boosting**. Liquidity is deposited into a Vault and the resulting ERC4626 are redirected into Ithil's boosting Service. Harvesters (see 2.6.2) periodically transfer part of the Vault's accrued fees into the Service, and if ERC4626 collateralization decreases, harvesters restore the boosters' total value by minting extra Vault's tokens (this means that the boosters are *senior* with respect to regular LP, and they are accepting a lower APY in return of a lower risk). When the agreement is exited, the relative ERC4626 tokens are redeemed.

- **Insurance**. Liquidity is deposited into a Vault and the resulting ERC4626 are redirected into Ithil's insurance Service. Harvesters (see 2.6.2) periodically transfer part of the Vault's accrued fees (a larger part than the Boosting case) into the Service, and if ERC4626 collateralization decreases, harvesters restore the Vault's total value by burning some Vault's tokens (this means that the insurers are *junior* with respect to regular LP, and they are accepting a higher risk in return of a higher APY). When the agreement is exited, the relative ERC4626 tokens are redeemed.

- **LP'ing**. Regular LP'ing can be considered a "trivial" Credit Service, with intermediate seniority between Boosters (senior) and Insurers (junior).

- **ITHIL call options**. Liquidity is deposited into a Vault and the resulting ERC4626 are redirected into Ithil's call option Service. This gives the right to buy a certain amount of ITHIL tokens at a given price (the *strike*) at a certain moment in the future (the *expiration date*). Both the strike and the expiration date are algorithmically computed. At the exit, the user can choose whether to buy or not the tokens, and depending on this choice either ITHIL or the original Vault's token is transferred to the user. This is an example of a Service with *multiple exit scenarios*.

- **ITHIL put options**. As in the previous case, but now the user has the right to *sell* ITHIL tokens at a given price (the *strike*) at a certain moment in the future (the *expiration date*). Both the strike and the expiration date are algorithmically computed.

4 The Vaults

We now look more deeply into the technical specifications of Ithil's Vaults. As mentioned in 2.1, the Vault is yield bearing token following the ERC4626 standard, and its purpose is collecting all liquidity deposited by LP's and Credit Services, and lending this liquidity to Debit Services.

First of all, the Vault inherits from ERC4626 and from OpenZeppelin's Ownable contract:

```
contract Vault is ERC4626, Ownable {...}
```

Since every Vault is deployed by the Manager contract (see 5), this contract is the owner of all the Vaults. We will not dig down into ERC4626 and Ownable's contract specification and redirect the interested reader into the respective contracts' documentations.

We will instead talk about Ithil's overrides to the inherited ERC4626 functions and Ithil's specific functions.

4.1 Accounting

The Vault keeps three state variables for accounting:

```
uint256 netLoans, uint256 currentProfits,
uint256 currentLosses, uint256 latestRepay
```

The variable `netLoans` registers the total amount of loans given across all Debit Services, while `currentProfits` and `currentLosses` are the total *locked* profits and losses respectively, at the moment of the latest repay. The block's timestamp of the latest repay is registered into `latestRepay`.

4.2 Locking

Fees and losses (see 4.6) undergo a locking period to dampen the movements of the Vault's token price per share and protect the Vault from flashloan attacks. The locked fees are computed via the `calculateLockedProfits()` function, whose formula is

$$L = P \cdot \frac{(T_L + T_U - T)_+}{T_U}$$

where L is the result of the function, P is the `currentProfits`, T_U is the Vault's `unlockTime`, a governance-decided variable initialized at 6 hours, T_L is the `latestRepay` and T is the current block's timestamp. The subscript "+" indicates the positive part.

In particular, when $T = T_L$, that is at the moment of profit (or loss) generation, we have $L = P$ and all `currentProfits` are locked. After the locking period, that is $T \geq T_L + T_U$, we will have $L = 0$ so all fees are unlocked. For $T_L \leq T \leq T_L + T_U$, we have

$$L = P \cdot \left(1 - \frac{T - T_L}{T_U}\right)$$

meaning that the fees and the losses unlock *linearly* with time.

The function `calculateLockedLosses()` does the exact same with the variable `currentLosses` instead.

4.3 Total Assets

Both borrowing from the Vault and repaying would modify the value of `totalAssets` as defined in the original ERC4626 contract and the resulting token price per share. Since loans are just temporarily moved from the vault, and since fees need to undergo a locking period, we need to factor this in into the definition of total assets. Therefore we have

$$\texttt{totalAssets} = \texttt{super.totalAssets + netLoans +}$$

$$\texttt{+ calculateLockedLosses() - calculateLockedProfits()}$$

with the use of mathematical overflow and capping checks so that to always make `totalAssets` > 0 and never make the above computation overflow (this is the recommended ERC4626 standard for overriding this function).

4.4 Free Liquidity

The Free Liquidity is the amount of liquidity that can be freely borrowed or withdrawn. Of course, it would be impossible to transfer an amount higher than the Vault's native token balance, but also locked profits should not be available for borrowing or withdrawal. Therefore, letting $L =$ `calculateLockedProfits()` we define

$$\texttt{freeLiquidity} = \begin{cases} \texttt{native.balanceOf(vault)} - L & \text{if } L > 0 \\ \texttt{native.balanceOf(vault)} & \text{otherwise.} \end{cases}$$

Notice that locked losses are not added to this calculation, since that would give the possibility of borrowing lost assets, thus breaking the mathematical soundness of the Vault.

An important caveat is that *the entirety of the free liquidity cannot be borrowed or withdrawn at once.* Indeed, this could cause the Vault to become **unhealthy** as per ERC4626 standard. Therefore, the functions `borrow`, `withdraw` and `redeem` have a check so to revert if the entirety of the free liquidity is taken.

4.5 Borrow

The Vault's `borrow(amount, receiver)` function is an only-owner function which directly transfers `amount` of the native asset from the Vault to a `receiver`. In order to register the loan, the `netLoans` state variable is incremented by `amount`. In practice, the receiver will be one of the whitelisted Debit Services, as the Router is in control of the list of receivers.

In order to insure the Vault is *healthy* as per ERC4626 standard (this means that the Vault's balance cannot be zero if the Vault's supply is higher than zero), the `amount` must be *strictly less* than the Vault's *free liquidity.*

The typical use case is liquidity lent to a Debit Service contract in order to make a user enter an Agreement.

By checking the formulas for `totalAssets()`, we see that the `borrow` function does not modify the total assets immediately after being called: the total assets are an *invariant* of the borrow function.

4.6 Repay

The Vault's `repay(assets, debt, repayer)` function is an only-owner function which directly transfers `assets` of the native asset from the `repayer` to the Vault. The `debt` is the amount of loan this function is declared to repay: the `netLoans` state variable is decremented by `debt` within this function.

The parameter `assets` can be either larger or smaller than `debt`. In the former case, we say that the difference `assets - debt` is the **fees** paid to the Vault for lending its liquidity; in the latter case, we say that the difference `debt - assets` is the **loss** incurred by the Vault for lending its liquidity. We call the former case a **Good Repay Event (GRE)** and the latter case a **Bad Repay Event (BRE)**.

Both the fees and the losses undergo a **locking period** which protects the Vault from flashloaners draining all the incoming fees and disincentivize LPs to frontrun losses. In order to do this, the `currentProfits` and `latestRepay` variables are updated as

$$\texttt{currentProfits} \mapsto \texttt{calculateLockedProfits() + (assets - debt)}_{+}$$

$$\texttt{currentLosses} \mapsto \texttt{calculateLockedLosses() + (debt - assets)}_{+}$$

$$\texttt{latestRepay} \mapsto \texttt{block.timestamp}$$

where the "+" subscript stands for the positive part of the subtraction.

Notice that `currentProfits` increases in a GRE, and the `currentLosses` increases in a BRE. Since the Vault's total assets are the sum of profits and losses, good repays and bad repays can compensate each other.

As per ERC4626 standards, a high `currentProfits` will tend to increase the Vault's token price per share, while a high `currentLosses` will tend to decrease it.

The typical use case is to repay a loan when exiting an Agreement of a Debit Service contract.

By checking the formulas for `totalAssets()` and locking, we see that the `repay` function does not modify the total assets immediately after being called: the total assets are an *invariant* of the repay function.

4.7 Direct mint

The Vault's `directMint(shares,receiver)` function is an only-owner function which directly mints `shares` of the Vault's ERC4626 tokens to the `receiver`. In practice, the receiver will be one of the whitelisted Credit Services, as the Manager is in control of the list of receivers.

This function has the effect of decreasing the Vault's price per share, while distributing part of the Vault's assets to the receiver by reflection. In that sense, it is considered a *loss* for the Vault and as such it needs to undergo the usual locking period. In order to calculate the loss, we use the native ERC4626 `convertToAssets` function and update

$$\texttt{currentProfits} \mapsto \texttt{calculateLockedProfits()}$$

$$\texttt{currentLosses} \mapsto \texttt{calculateLockedLosses() + convertToAssets(shares)}$$

$$\texttt{latestRepay} \mapsto \texttt{block.timestamp}$$

The typical use case of this function is the distribution of part of the accrued fees to a Credit Service.

By checking the formulas for `totalAssets()` and ignoring unlocking, we see that the `directMint` function decreases the total assets immediately after being called. However, as long as currentProfits are positive, it is always possible to choose a value for `shares` such that unlocking of profits compensate the distribution of fees to the receiver, so that the other lenders do not experience any decrease in their share value.

4.8 Direct burn

The Vault's `directBurn(shares,owner)` function is an only-owner function which burns `shares` of the Vault's ERC4626 tokens to the `owner`. In practice, the owner will be one of the whitelisted Credit Services, as the Manager is in control of the list of owners. In order for the burning to be successful, the owner must have approved the Vault to burn at least the amount of shares to be burned.

This function has the effect of increasing the Vault's price per share, while distributing part of the owner's assets to the Vault by reflection. In that sense, it is considered a *profit* for the Vault and as such it needs to undergo the usual locking period. In order to calculate the profit, we use the native ERC4626 `convertToAssets` function and update

$$\texttt{currentProfits} \mapsto \texttt{calculateLockedProfits() + convertToAssets(shares)}$$

$$\texttt{currentLosses} \mapsto \texttt{calculateLockedLosses()}$$

$$\texttt{latestRepay} \mapsto \texttt{block.timestamp}$$

The typical use case of this function is the return of part of the accrued fees from a Credit Service to the Vault. By checking the formulas for `totalAssets()` and ignoring unlocking, we see that the `directBurn` function increases the total assets immediately after being called. The same unlocking reasoning as in `directMint` apply to this function.

5 Manager

The **Manager** is the contract in charge of managing cashflows and risk across the Vaults and the Services. The Manager is the also the deployer and owner of all the Vaults (in particular, it can call the borrow, repay, directMint and directBurn functions of the Vaults). By setting a nonzero **cap**, the Manager can whitelist a given Service, while setting the **risk spreads** the Manager defines the risk of each Service. In this way, Ithil can provide a complete risk management system easily scalable and with a full control of the various allocations.

The core of the Manager thus consists of two state variables:

```
mapping(address => address) vaults
```

```
mapping(address => mapping(address => RiskParams)) riskParams
```

where `RiskParams` is a data structure containing the spreads and caps of each Service:

```
struct RiskParams = {uint256 cap, uint256 riskSpread}
```

The `vaults` mapping simply contains the Vault address for each underlying asset, while the `riskParams` register the risk parameters of each Service address (first argument) for a given token (second argument).

5.1 Manager's functions

The Manager's function fall into two categories: onlyOwner functions and whitelisted functions.

The **onlyOwner** functions can only be used by Ithil's governance and are the following:

- `create(address token)`: deploys a new Vault with `token` as underlying asset
- `setSpread(address service, address token, address spread)`: sets a spread in `riskParams` for a given service and token
- `setCap(address service, address token, address spread)`: sets a cap in `riskParams` for a given service and token

The **whitelisted** functions can only be called by addresses whose `cap` in `riskParams` is positive.

- `borrow(token, amount, exposure, receiver)`: calls the Vault's `borrow` function and checks whether the new exposure (see 6) stays below the Service's cap for the particular `token`. In the parameters, `exposure` is the current exposure of the Service and it is stored in the Service itself and passed to the Manager. Since Services are whitelisted by the Governance (by setting a nonzero cap), we can assume the datum passed to the Manager as exposure is reliable.
- `repay(token, amount, debt, repaier)` calls the Vault's `repay` function (no exposure checks are performed on this function).
- `directMint(token,to,shares,exposure)` calls the Vault's `directMint` funciton and checks whether the new exposure (see 6) stays below the Service's cap for the particular `token`.
- `directBurn(token,from,shares,exposure)` calls the Vault's `directBurn` funciton and checks whether the new exposure (see 6) stays below the Service's cap for the particular `token`.

6 Services

The Manager allows any address with positive cap to borrow and repay the Vaults. Therefore, in theory, any address could be whitelisted. In practice, the Governance whitelists other Smart Contracts called **Services**.

In principle, anybody can suggest a Service and submit it for approval by the Governance. Currently, Ithil developers have designed a suite of Smart Contracts, aiming to make the development of new Services as easy as possible, yet producing well-thought and secure ways to use the Vaults' liquidity. In this section, we describe this suite, while in the next section we discuss a few examples about how to apply this architecture in practice.

It is important to stress out that Ithil's core, consisting of the Manager and the Vaults, is agnostic of the functioning of the whitelisted contracts, therefore (as long as the Governance agrees), even a Service which does not respect the structure described below might be whitelisted in the future.

6.1 Base Service

The **Base Service** contract (also simply called `Service`) is an abstract contract containing the basic functionality every Service must have in order to be whitelisted. The contract inheritance is as follows:

```
abstract contract Service is Ownable, ERC721 {...
```

In particular, it is minted as an NFT to the caller or the `open` function. We call *user* the caller of this function, and we say that the user *enters an Agreement*.

6.1.1 Storage and data structures

The main storage of the Service consists of two state variables:

```
mapping(address => uint256) exposures
Agreement[] agreements
```

The `exposures` variable registers the sum of all loan amounts for a given token. This variable is passed to the Manager (see 5) to enforce caps.

The `agreements` is an array of data structures:

```
struct Agreement = {Loan[] loans, Collateral[] collaterals,
    uint256 createdAt, Status status}
```

which in turn contains the data structures

```
struct Loan = {address token, uint256 amount,
    uint256 margin, uint256 interestAndSpread}
```

and

```
struct Collateral = {ItemType item, address token,
    uint256 identifier, uint256 amount}.
```

The `Loan` structure contains the data of the assets flow *to or from the Vault* [2] for a given token, registering the eventual margin posted by the user, the interest rate and risk spread applied (these two are packed in a single integer to save gas).

The `Collateral` structure contains the data of the assets *locked as collateral* to the Loan given or taken. This specifies the `ItemType`, which currently can be ERC20, ERC721 or ERC1155, the token address, the identifier (only applicable for the ERC721 item type) and the amount.

[2] In case of flow towards the Vault, the lender of the loan is the user.

6.1.2 Functions

The public functions of the Service contract are all `virtual` and overridden by the inheriting Contracts, and they are the following:

- `open(Order order)`: mints an NFT to the user, performs an internal abstract `_open` function, implemented on a derived contract (see later subsections), and stores the resulting Agreement in the `agreements` with a particular index. The `Order` is a data structure containing the desired Agreement together with additional `data` to adjust to the particular implementation of the Service.

- `close(uint256 index, bytes data)`: burns the relative NFT from the user, performs an internal abstract `_close` function, and deletes the corresponding Agreement from the `agreements`. The `data` paarameter allows to adjust to the particular implementation of the Service.

- `edit(uint256 index, Agreement agreement, bytes data)`: allows the owner of the NFT to change the existing Agreement in the `agreements` into the new one `agreement`. The `data` parameter allows to adjust to the particular implementation of the Service. Currently it is fully virtual, not implemented in the Base Service.

6.2 Debit Service

The **Debit Service** is the prototype of a Smart Contract calling (through the Manager) the `borrow` and `repay` functions of the Vaults. Since it still does not have a precise implementation, the contract definition is as follows:

```
abstract contract DebitService is Service {...
```

It does not have any native storage (beyond the one inherited by the Service contract). The functions are both contract specific and overridden ones.

The **overridden function** are as follows:

- `open(Order order)`: updates the `exposures` by adding the Loan's `amount`, transfers the Loan's `margin` from the caller (in particular, the caller must approve the contract beforehand), and borrows the Loan's `amount` by calling the Manager's `borrow` function. Finally, it calls the overlying `open` function.

- `close(uint256 index, bytes data)`: calls the overlying `close` function, updates the exposures by subtracting the loan amounts, and repays the Vault by calling the Manager's `repay` function. A modifier enforces that the caller must be the owner of the NFT with that index, unless the Agreement is liquidable (see below).

The **specific functions** take into account the *default risk* of the borrowing procedure, which is mitigated through liquidation:

- `quote(Agreement agreement)`, a virtual function without implementation, is aimed to compute the amount of tokens obtained in a `close` call done in that specific moment.

- `liquidationScore(uint256 id)` a virtual function computing a number which determines the "health" of the position by considering the Loan of `agreements[id]` and calling the `quote` function. By default, the Agreement is *liquidable* if its liquidation score is positive: in this case, anybody can close the Agreement by calling the `close` function. The precise calculation uses the **liquidation threshold**:

$$LT = L + M \cdot \frac{S}{I + S}$$

where L is the loan amount, M is the margin, S is the risk spread and I is the interest rate applied to that particular agreement. The liquidation score is then

$$LS = \left(1 - \frac{Q}{LT}\right)_+$$

where Q is the quoted amount. [3] This formula is valid for a single token loan: the liquidation score of a multi-token loan is the sum of the single liquidation scores for each token.

6.3 Securitisable Service

In traditional finance, **securitisation** is the process of selling a particular asset or basket of assets under management to a willing investor who takes their risk and rewards.

In our implementation, a **Securitisable Service** is a Debit Service whose credit, which initially belongs to the Vault, can be freely *purchased* by directly repaying the Vault at a discount. This has actually the effect of changing the *lender*: that's why the contract has a storage state variable

$$mapping(uint256 => address) lenders$$

who associates, for each index, the address of the lender who purchased the credit.

The Contract definition is defined as

```
abstract contract SecuritisableService is DebitService {...
```

and it consists of:

- The specific function `purchaseCredit(uint256 id)`, which computes the **fair price** of the credit coming from the Agreement with `id` index and repays the Vault via a call to the Manager's `repay` function and stores the purchaser into the `lenders` storage variable. It also decreases the `exposures` accordingly. The fair price is calculated as

$$FP = L + F \cdot \frac{S}{I + S}$$

where L is the loan amount, F are the fees accrued so far, S is the risk spread and I is the interest rate applied to that particular agreement.

- The overridden function `close` checks whether a given lender has been initialized: in this case, the repay is done to the lender via a direct transfer, rather than through a `repay` call.

[3] In order to avoid rounding and normalization errors, this quantity is a dimensionless 18 digit fixed point integer.

6.4 Credit Service

The **Credit Service** is the prototype of a Smart Contract calling (through the Manager) the `directMint` and `directBurn` functions of the Vault. In order to be able to have this privilege, the user must deposit some liquidity to the Vault, therefore these services can be considered as a loan given *by the user to the Vault* (that is, the user acquires a credit, rather than a debit, towards the Vault).

The definition of this contract is

```
abstract contract CreditService is Service {...
```

and it only consists of two overridden functions:

- `open` calls the overlying open function, calls the `deposit` ERC4626 function of the Vault and updates the `exposures` with the obtained shares.
- `close` calls the overlying close function and updates the `exposures`.

Since the exit from credit Agreements can be of various types (see the *multiple exit scenarios* in 3.2), we do not assume a `withdraw` is called at exit, and leave the specific implementaiton to the inheriting contracts.

6.5 Whitelisted Service

In **Whitelisted Services**, the Governance maintains a mapping

```
mapping(address=>bool) whitelisted
```

which checks whether a given user is whitelisted or not. Only whitelisted users can access to such services if the `enabled` flag is set to true, and this is enforced by implementing the `_beforeOpening` hook. The definition of this contract is

```
abstract contract WhitelistedService is Service {...
```

in particular, implementations of any of the types of services described above (Debit, Credit, Securitised) may or may not inherit from Whitelisted.

7 Fees

7.1 Payoffs

7.2 Interest rate

7.3 Risk spread

8 Harvesting

8.1 Credit Services

8.2 Debit Services