Decentralized Vault Portfolios

Christian Schmitz   Pablo Antonio Bejarano

13th July 2024

Abstract

This article introduces Decentralized Vault Portfolios (DVPs). DVPs are **tokenized investment funds** controlled by a smart contract, tailored to extended Unspent Transaction Output (eUTxO) blockchains. DVPs allow management of funds without custody, offering unparalleled transparency and security to their token holders.

Contents

1 Introduction **4**

2 Tokenization **5**

3 Vault **6**

4 Orders **7**

5 Price **8**

6 Fees **10**

7 Governance **17**

8 Oracles **19**

9 Staking **19**

10 Compliance **20**

A Tokens and datums **21**

B Validators **25**

C Transactions **45**

List of Terms

ADA	Cardano's native cryptocurrency
address	blockchain analogy of a bank account number
datum	information attached to an Unspent Transaction Output (UTxO)
lovelace	one millionth of an ADA
validator	an on-chain script that validates a transaction
witness	a public key or a validator script. A transaction has at least one witness

Acronyms

AML	Anti Money Laundering
CFT	Combating the Financing of Terrorism
CIP	Cardano Improvement Proposal
DAG	Directed Acyclical Graph
DAO	Decentralized Autonomous Organization
DVP	Decentralized Vault Portfolio
eUTxO	extended Unspent Transaction Output
KYC	Known Your Customer
NFT	Non-Fungible Token
UTxO	Unspent Transaction Output

Mathematical symbols

Δ_α	success fee dilution
Δ_μ	management fee dilution
N	total number of DVP tokens in circulation
V	total asset value of a DVP
α	DVP token success, a ratio of two benchmark prices
δ_α	provisional success fee
δ_b	burn fee, as a number of tokens
$\hat{\delta}_b$	min burn fee, as a number of tokens
δ_m	mint fee, as a number of tokens
$\hat{\delta}_m$	min mint fee, as a number of tokens
ϕ_α	relative success fee, a function
ϕ_b	relative burn fee
ϕ_m	relative mint fee
ϕ_μ	relative management fee
n_b	number of DVP tokens in a burn order
p	on-chain DVP token price relative to ADA
π	on-chain DVP token price relative to a benchmark

π_{ref}	start of period on-chain DVP token price relative to a benchmark
π^*	on-chain DVP token price relative to a benchmark, after the success fee is charged
t_{tx}^-	start of transaction validity timerange interval
t_{tx}^+	end of transaction validity timerange interval
v	value of deposited or withdrawn assets

1 Introduction

Blockchains and smart contracts are novel technologies that can radically increase the transparency and security of financial services. Decentralized Vault Portfolios (DVPs) are t**okenized investment funds** that take advantage of these innovations.

DVPs have the following notable properties:

i. Tokenization of fund shares

ii. Active or passive management without custody

iii. Entry or exit at any time, anywhere (instant global liquidity)

iv. Participation of any size

v. Conventional fee structure (entry/exit fee, management fee, success fee)

vi. Large investment universe (hundreds of assets per DVP)

vii. Eliminates parasitic management practices (e.g. charging commission on trades)

viii. Decentralized control of parameters

ix. Compliant (AML/CFT)

Many of these properties naturally favor extended Unspent Transaction Output (eUTxO) blockchains and we have chosen to develop the first implementation of the DVP smart contract for **Cardano** (an advanced public eUTxO blockchain).

Some concepts used in this article are Cardano-specific and might not yet have equivalents in other eUTxO blockchains.

Examples

Blue boxes contain calculation examples with concrete values.

Notes

Orange boxes contain security notes that deserve special attention.

2 Tokenization

By representing DVP shares as cryptocurrency tokens, global exposure to a DVP is available through secondary markets. Tokenization also allows DVP shares to be used for payments, similar to cash, and to interact with other smart contracts (e.g. an inheritance smart contract).

Henceforth, we will refer to DVP shares as *DVP tokens*.

Example: simple fee-less tokenized fund

Imagine a fund initially composed as follows:

Asset	**Quantity**	**Unit price** [USD]	**Value** [USD]
USD	50	1	50
ADA	100	0.5	50
DVP token	100	1	100

The 100 DVP tokens minted upon formation each have an initial value of 1 USD. The holder of such a token can withdraw 1 USD-worth of assets from the fund per token. For example a user with 10 tokens can withdraw 5 USD + 10 ADA. After such a withdrawal 10 tokens are taken out of circulation and the fund composition becomes:

Asset	**Quantity**	**Unit price** [USD]	**Value** [USD]
USD	45	1	45
ADA	90	0.5	45
DVP token	90	1	90

If the ADA price suddenly doubles, from 0.5 USD to 1 USD, the composition changes to:

Asset	**Quantity**	**Unit price** [USD]	**Value** [USD]
USD	45	1	45
ADA	90	1	90
DVP token	90	1.5	135

A user entering the fund after such a market event would have to deposit 1.5 USD-worth of assets per DVP token received. For example a deposit of 15 USD results in 10 DVP tokens returned, changing the fund composition:

Asset	**Quantity**	**Unit price** [USD]	**Value** [USD]
USD	60	1	60
ADA	90	1	90
DVP token	100	1.5	150

3 Vault

The DVP smart contract protects funds locked at the vault address, ensuring the value of the DVP tokens is maintained. The following three transaction types interact with the vault address:

1. Swap
2. Withdraw
3. Deposit

3.1 Swap

Swapping vault assets must conserve (or increase) the total value locked at the vault address. The DVP also ensures that the various assets are correctly counted. These counts facilitate the calculation of the on-chain token price.

An on-chain oracle price feed is used when calculating changes to the total value locked at the vault address.

The use of oracles is what allows **active management without custody**. The oracle configuration is of critical importance to vault security.

Liquidity provision

By spreading the assets over any number of UTxOs, the swap transaction can be used to provide liquidity to multiple other protocols in parallel.

Note: only one asset class per UTxO

By permitting only one asset class per vault UTxO (in addition to ADA) we avoid a UTxO spam exploit, which make such UTxOs unspendable. More generally we need to ensure that internal DVP UTxOs can never contain more than a few asset classes.

3.2 Withdraw

The net value of the withdrawn assets is calculated using the same on-chain oracle price feeds. This value must not be greater than the value of the DVP tokens being burned in the same transaction.

3.3 Deposit

The net value of the deposited assets results in a number of DVP tokens of equal value being minted and returned to the users.

4 Orders

Users interact with a DVP by submitting orders. There are two types of orders:

1. Mint orders (requests to deposit funds and receive DVP tokens in return)
2. Burn orders (requests to withdraw funds by exchanging DVP tokens)

A user must specify the following details for each order (contained in the datum):

- Return address (wallet or another contract)
- Return datum (should be unique to prevent the double satisfaction exploit)
- Minimum received tokens in case of a mint order, minimum received value in case of a burn order
- Maximum age of the on-chain token price and asset prices involved in the transaction

The minimum return value must be set to realistically, taking into account the fees. Unfavorable or badly configured orders are simply ignored by the DVP manager.

Mint orders are sent to the mint order validator, and burn orders are sent to the burn order validator. These validators act in the interest of the user.

Each order is submitted as a separate UTxO. Once an order is submitted two actions are possible:

1. The order can be canceled by the user
2. The order can be fulfilled by the DVP manager

An order can sit indefinitely at the address of the mint or burn order validator if the minimum received value condition can't be met and the user decides not to cancel.

Note: datum tagging to prevent double satisfaction

Upon order fulfillment, the returned UTxO must have the requested datum. Without datum tagging the smart contract would allow two orders with the same return address to be fulfilled at the same time by only returning sufficient value for the largest of the two orders.

5 Price

An important aspect of DVPs is the on-chain calculation of the DVP token price. Having this price available on-chain has several advantages:

- No reliance on secondary market arbitrageurs (initially secondary markets will be very illiquid and oracle price feeds for the DVP token won't be accurate)
- The success fee can be calculated deterministically
- The mint and burn order validators can guarantee a fair conversion at all times

The on-chain token price must be updated frequently, a non-trivial action. DVPs do this by keeping track of each underlying asset using counters. An on-chain token price update then proceeds by summing over these asset counters, combined with oracle price feeds, to calculate the ratio of the total asset value and the token circulating supply.

5.1 Initial price

The initial on-chain price of a DVP token can be arbitrarily set. The initial price determines how many tokens are minted by the initial deposit.

5.2 Asset counting

The number of tokens of each underlying asset in the vault is tracked on-chain using special counters. The set of these asset counters must match exactly the intended investment universe at all times. Duplicate asset counters aren't possible.

Note: duplicate asset counters

If duplicates were possible one of the duplicates could contain a low count, the other a high count, and alternating between the two when calculating the total value would allow the DVP manager to mint at a low price and burn at a high price, ignoring any other orders while taking advantage of this exploit.

Adding/removing asset counters

Initially the investment universe only contains ADA.

Before adding a new asset counter, a proof must be generated that it doesn't already exist. Generating this proof might require many transactions, as it involves iterating over all the asset counters in existence.

Finally the proof is used when adding the asset counter for the given asset class.

Removing an asset class requires an existence proof to be generated for the given asset class, and that its count is 0.

5.3 Price update

The on-chain DVP token price is the ratio of the total fund value over the number of DVP tokens in circulation. This must be updated regularly to reflect changes in the composition of the fund, and to reflect changes of the price changes of the underlying assets.

The asset counters can't be changed during this process, which can be ensured by making sure they are older than the initiation of the total asset value calculation.

Timestamps are not very accurate when working with deterministic transactions because they can only be compared to intervals. Instead of using time directly, DVPs increment an on-chain *tick* value.

This *tick* is incremented every time an asset counter is updated.

The total asset value calculation can then start by copying the ADA count and the current *tick* value. All subsequent asset counters being iterated over must then have a *tick* that is smaller or equal to the start *tick*.

The final price update ensures all asset counters have been iterated over, and copies the total asset value into the `price` UTxO.

Price recency

The *tick* allows us to chain price update transactions. However, we still need to use timestamps to ensure prices are recent. During total asset value calculation, the DVP token price timestamp t_p is taken as the minimum of the asset price timestamps:

$$t_p = \min(t_a) \quad \forall a \in A_{vault}$$

We then compare t_p to the end of the transaction validity time-range t^+_{tx} to ensure the price is more recent than a period τ_p:

$$t_p \geq t^+_{tx} - \tau_p$$

6 Fees

DVPs can be configured with the following fees:

1. Entry fee
2. Exit fee
3. Management fee
4. Success fee

Each fee is optional, and charged in the form of DVP tokens.

6.1 Entry fee

The entry fee, also referred to as the mint fee, is a constant fraction of the deposited value, bound by a minimum.

This lower bound ensures that at least the blockchain network fees are covered by the mint fee.

Mint fee formula

Let p denote the current DVP price in ADA, let v denote the equivalent ADA value of the order, let ϕ_m denote the relative mint fee, and let $\hat{\delta}_m$ denote the mint fee lower bound. The number of DVP tokens withheld as a mint fee δ_m is calculated as:

$$\delta_m = \max(\phi_m \cdot \frac{v}{p},\ \hat{\delta}_m)$$

Example: mint fee calculation

Assume the DVP charges a mint fee of 0.5%, with a minimum of 0.02 tokens, and that the current on-chain DVP token price is 100 ADA/token. A deposit of 200 ADA results in 2 DVP tokens being minted. The number of DVP tokens withheld as a mint fee is calculated as:

$$0.005 \cdot \frac{200}{100} = 0.005 \cdot 2 = 0.01$$

Because $0.01 < 0.02$, the mint fee is corrected upward to **0.02**.

6.2 Exit fee

The exit fee is a combination of a burn fee and a provisional success fee. This fee is thus a non-constant fraction of the tokens burned n_b.

Burn fee formula

The burn fee is calculated like the mint fee, parametrized with its own fraction and lower bound:

$$\delta_b = \max(\phi_b \cdot (n_b - \delta_\alpha),\ \hat{\delta}_b)$$

The provisional success fee δ_α is explained in section 6.4.

6.3 Management fee

The management fee is charged for all DVP token holders by diluting the DVP token supply by a constant fraction. The DVP ensures that this cannot happen more frequently than a configured period (typically 24 hours) by validating the change of the last management fee mint event t_μ:

$$t_{\mu,1} \geq t_{tx}^{+}$$
$$t_{\mu,0} \leq t_{tx}^{-} - \tau_\mu$$

DVP token dilution when charging the management fee decreases the value of each DVP token.

Note: why the management fee can't be accumulated

The management fee is charged for all token holders in the form of the token dilution. Token dilution has an immediate impact on the on-chain token price. Dilution events should be as predictable as possible to avoid influencing secondary markets. If it would be possible to accumulate the management fee over many days, a malicious DVP manager could borrow a large amount of DVP tokens, sell them, suddenly negatively impact the price by minting all the management fee at once, and buy the DVP tokens back at a cheaper rate (i.e. shorting the DVP tokens with leverage, taking advantage of insider information).

Management fee formula

Let N denote the number of DVP tokens in circulation, and let V denote the DVP total asset value. If we want to charge a periodic fee of ϕ_μ (typically 0.01% daily, or 3.65% annually), as a number of DVP tokens equivalent in value to $\phi_\mu \cdot V$, the original N tokens will be worth only $(1 - \phi_\mu) \cdot V$.

Using the equivalence of each DVP token, we can calculate the management fee dilution Δ_μ as:

$$\frac{\phi_\mu \cdot V}{\Delta_\mu} = \frac{(1 - \phi_\mu) \cdot V}{N}$$
$$\Rightarrow\ \Delta_\mu = N \cdot \frac{\phi_\mu}{1 - \phi_\mu}$$

Note that V doesn't impact this calculation.

Example: management fee calculation

Assume the DVP charges a daily management fee of 0.01%, and there are currently 1000 DVP tokens in circulation. The DVP manager is allowed to mint the following number of DVP tokens as a management fee once the day has passed:

$$1000 \cdot \frac{0.0001}{1 - 0.0001} = 1000 \cdot \frac{0.0001}{0.9999} \approx 0.10001$$

The resulting number is very close to $\phi_\mu \cdot N = 0.1$. For larger relative management fees the difference will be more pronounced.

6.4 Success fee

Many conventional funds charge an annual success fee depending on their performance with respect to some benchmark (eg. S&P 500).

DVPs can be configured with a progressive success fee, which can approximate any mathematical function.

Success fees are an important incentive for DVP managers to react quickly to market changes and to properly analyze and model the future performance of the underlying assets.

Success fees are more complex than daily management fees, because all token holders can't be diluted equally. Token holders that minted at a higher token price should be charged less than users who held the tokens since before the start of the year. We accomplish this by diluting all holders equally at the end of the year, and then partially reimbursing holders who have some proof they minted tokens during the year at a price level higher than the start of the year. We refer to these proofs as **vouchers**.

Benchmark

DVPs might prefer benchmarks other than comparing to the blockchain's base currency, and will thus need to combine the on-chain DVP token price with an oracle price feed for that particular benchmark.

Success fee formula

Let π_{ref} and π denote the on-chain year-start and year-end prices relative to the benchmark respectively. Success α is calculated as:

$$\alpha = \frac{\pi}{\pi_{ref}}$$

If $\alpha \leq 1$, no success fee is charged. If $\alpha > 1$, a fraction of $\alpha - 1$ is charged as the relative success fee. Let ϕ_α denote this fraction, which is a progressive step function with coefficients c_i and steps σ_i:

$$\phi_\alpha(\alpha) = \sum_{i=0}^{k-1} \Big(c_i \cdot \min[\max(\alpha - \sigma_i, 0), \sigma_{i+1} - \sigma_i] \Big) + c_k \cdot \max(\alpha - \sigma_k, 0) \qquad \text{with } \sigma_0 = 1$$

$\phi_\alpha(\alpha)$ is visualized in figure 1.

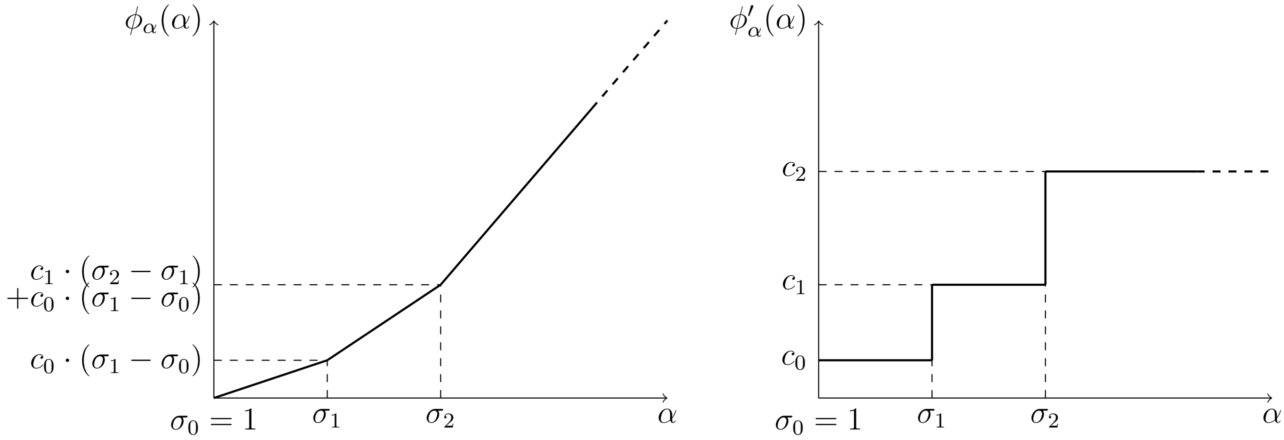


Figure 1: a progressive step function with two steps, and its derivative

Upon dilution the token holders expect to keep $\alpha - \phi_\alpha(\alpha)$ of the success. Let α^* denote this net success after dilution. The price after dilution π^* is calculated as:

$$\alpha^* = \frac{\pi^*}{\pi_{ref}} \equiv \alpha - \phi_\alpha(\alpha) = \frac{\pi}{\pi_{ref}} - \phi_\alpha(\alpha)$$
$$\Rightarrow \ \pi^* = \pi - \pi_{ref} \cdot \phi_\alpha(\alpha)$$

Because total asset value V doesn't change, $\pi^* \cdot (N + \Delta_\alpha) = \pi \cdot N$ holds. The success fee dilution Δ_α is calculated as:

$$\pi^* = \pi - \pi_{ref} \cdot \phi_\alpha(\alpha) \equiv \frac{N \cdot \pi}{N + \Delta_\alpha}$$
$$\Rightarrow \ \Delta_\alpha = \frac{N \cdot \pi}{\pi - \pi_{ref} \cdot \phi_\alpha(\alpha)} - N$$
$$\Rightarrow \ \Delta_\alpha = \frac{N \cdot \alpha}{\alpha - \phi_\alpha(\alpha)} - N$$
$$\Rightarrow \ \boxed{\Delta_\alpha = N \cdot \frac{\phi_\alpha(\alpha)}{\alpha - \phi_\alpha(\alpha)}}$$

Note: valid ϕ_α coefficients

Monotonicity of c_i is not a requirement, but it is important that each coefficient lies within a limited range to avoid datum spam.

$$0 \leq c_i \leq 1 \quad \text{and} \quad \sigma_{i-1} < \sigma_i \leq \sigma_{max}$$

Example: success fee calculation

Assume the DVP charges a success fee of 30% on any success above 5% (so $\sigma_0 = 1$, $c_0 = 0$, $\sigma_1 = 1.05$ and $c_1 = 0.3$), there are 1000 DVP tokens in circulation, the benchmark is ADA, and the price increased from 100 ADA/token to 150 ADA/token. The success fee fraction ϕ_α is calculated as:

$$\begin{aligned} \alpha &= \frac{150}{100} = 1.5 \\ \phi_\alpha(1.5) &= 0 \cdot \min[\max(1.5 - 1, 0), 0.05] \\ &\quad + 0.3 \cdot \max(1.5 - 1.05, 0) \\ &= 0.3 \cdot 0.45 = 0.135 \end{aligned}$$

The success fee that can be minted at the end of the year is calculated as:

$$\Delta_\alpha = 1000 \cdot \frac{0.135}{1.5 - 0.135} \approx 98.901099$$

After the success fee is minted the on-chain DVP token price decreases to:

$$\begin{aligned} 1000 \cdot 150 &= (1000 + 98.901099) \cdot \pi^* \\ \Rightarrow \quad \pi^* &= 150 \cdot \frac{1000}{1098.901099} = 136.5 \text{ ADA/token} \end{aligned}$$

$136.5 = 150 - 13.5$, so this is the expected DVP token price after dilution.

Vouchers

Vouchers are on-chain proofs that a user minted a number of DVP tokens at a certain benchmark price level. A voucher is used to lower the provisional success fee when burning DVP tokens, or to receive an end-of-year success fee reimbursement.

A voucher consists of two CIP-68 NFTs:

1. A reference token locked at the voucher validator address
2. A user token returned to the user

Both NFTs have the same serial number. The user token will have a non-zero value before the year's success is fee minted, so during this time it might be traded on secondary markets.

Provisional success fee when burning

A user that would withdraw right before the end of the year wouldn't see an on-chain token price affected by the upcoming success fee dilution. To prevent users taking advantage of this, a provisional success fee must be charged as part of the exit fee.

The provisional success fee is calculated by comparing the most recent on-chain benchmark price with that of the start of the year.

Let n_b denote the number of tokens being burned and let π denote the on-chain benchmark price at the moment of burning. The provisional success fee δ_α is calculated as if the success fee would be minted right at that moment and the user would be left with the same number of tokens n_b at a lower price π^*:

$$
\begin{aligned}
\alpha &= \frac{\pi}{\pi_{ref}} \\
(n_b - \delta_\alpha) \cdot \pi &= n_b \cdot \pi^* \quad \text{and} \quad \pi^* = \pi - \pi_{ref} \cdot \phi_\alpha(\alpha) \\
\Rightarrow \; n_b - \delta_\alpha &= n_b \cdot \left(1 - \frac{\pi_{ref}}{\pi} \cdot \phi_\alpha(\alpha)\right) \\
\Rightarrow & \boxed{\delta_\alpha = n_b \cdot \frac{\phi_\alpha(\alpha)}{\alpha}}
\end{aligned}
$$

Example: provisional success fee calculation without vouchers

Assume the DVP charges a success fee of 30% on any success above 5%, the benchmark is ADA, the token price at the start of the year was 100 ADA/token, and the current token price is 140 ADA/token. A user burning 10 DVP tokens would be charged the following provisional success fee:

$$
\begin{aligned}
\alpha &= \frac{140}{100} = 1.4 \\
\phi_\alpha(1.4) &= 0.3 \cdot (1.4 - 1.05) = 0.105 \\
\delta_\alpha &= 100 \cdot \frac{0.105}{1.4} = 0.75 \text{ tokens}
\end{aligned}
$$

The 9.25 remaining tokens have a value of 1295 ADA, so the user sees an effective gain of 29.5%, instead of the internal 40% DVP gain. The difference is 10.5%-pt, as expected.

Including vouchers in the burn order allows calculating δ_α using a higher price for part of the n_b tokens. Let n_v denote the number of DVP tokens when the voucher was minted, and let π_v denote the benchmark price at time of minting. The provisional success fee δ_α is now calculated as:

$$
\begin{aligned}
\alpha_v &= \frac{\pi}{\pi_v} \\
\delta_\alpha &= \sum_{vouchers} \left(n_v \cdot \frac{\phi_\alpha(\alpha_v)}{\alpha_v}\right) + (n_b - \sum_{vouchers} n_v) \cdot \frac{\phi_\alpha(\alpha)}{\alpha}
\end{aligned}
$$

The provisional success fee can become negative if too many vouchers are included, in this case no provisional success fee is charged. Vouchers are always burned in their entirety, so such a burn order would be wasteful.

Example: provisional success fee calculation with vouchers

Again assume the DVP charges a success fee of 30% on any success above 5%, the benchmark is ADA, the token price at the start of the year was 100 ADA/token, and the current token price is 140 ADA/token. A user burning 10 DVP tokens and 1 voucher for 5 tokens at a price of 120 ADA/token would be charged the following provisional success fee:

$$
\begin{aligned}
\alpha &= \frac{140}{100} = 1.4 \\
\alpha_v &= \frac{140}{120} \approx 1.166667 \\
\phi_\alpha(1.4) &= 0.3 \cdot (1.4 - 1.05) = 0.105 \\
\phi_\alpha(1.166667) &= 0.3 \cdot (1.166667 - 1.05) = 0.035 \\
\delta_\alpha &= 5 \cdot \frac{0.035}{1.166667} + 5 \cdot \frac{0.105}{1.4} = 0.525 \text{ tokens}
\end{aligned}
$$

Success fee reimbursement

The number of tokens that would be charged from the start of the year until the current price level, minus the success fee charged from the price level of the voucher until the current on-chain price, must be reimbursed at the end of the year if the voucher wasn't used during the year.

The number of reimbursed tokens n_{reim} is calculated as:

$$
\alpha = \frac{\pi}{\pi_{ref}} \quad \text{and} \quad \alpha_v = \frac{\pi}{\pi_v}
$$

$$
n_{reim} = n_v \cdot \left(\frac{\phi_\alpha(\alpha)}{\alpha - \phi_\alpha(\alpha)} - \frac{\phi_\alpha(\alpha_v)}{\alpha_v - \phi_\alpha(\alpha_v)} \right)
$$

Example: success reimbursement calculation

Assume the DVP charges a success fee of 30% on any success above 5%, the benchmark is ADA, the token price at the start of the year was 100 ADA/token, and the year end token price is 150 ADA/token. A user has an outstanding voucher for 10 DVP tokens at a price level of 120 ADA/token. The reimbursement is calculated as:

$$
\begin{aligned}
\alpha &= \frac{150}{100} = 1.5 \\
\alpha_v &= \frac{150}{120} = 1.25 \\
\phi_\alpha(1.5) &= 0.3 \cdot (1.5 - 1.05) = 0.135 \\
\phi_\alpha(1.25) &= 0.3 \cdot (1.25 - 1.05) = 0.06 \\
n_{reim} &= 10 \cdot \left(\frac{0.135}{1.5 - 0.135} - \frac{0.06}{1.25 - 0.06} \right) \\
&= 10 \cdot (0.098901 - 0.050420) \approx 0.48481 \text{ tokens}
\end{aligned}
$$

7 Governance

DVPs have a set of updateable parameters, listed in table 5.

Every parameter update goes through the governance process visualized in figure 2. Parameter updates are delayed and can't be done concurrently. This process gives the maximal possible visibility to each update.

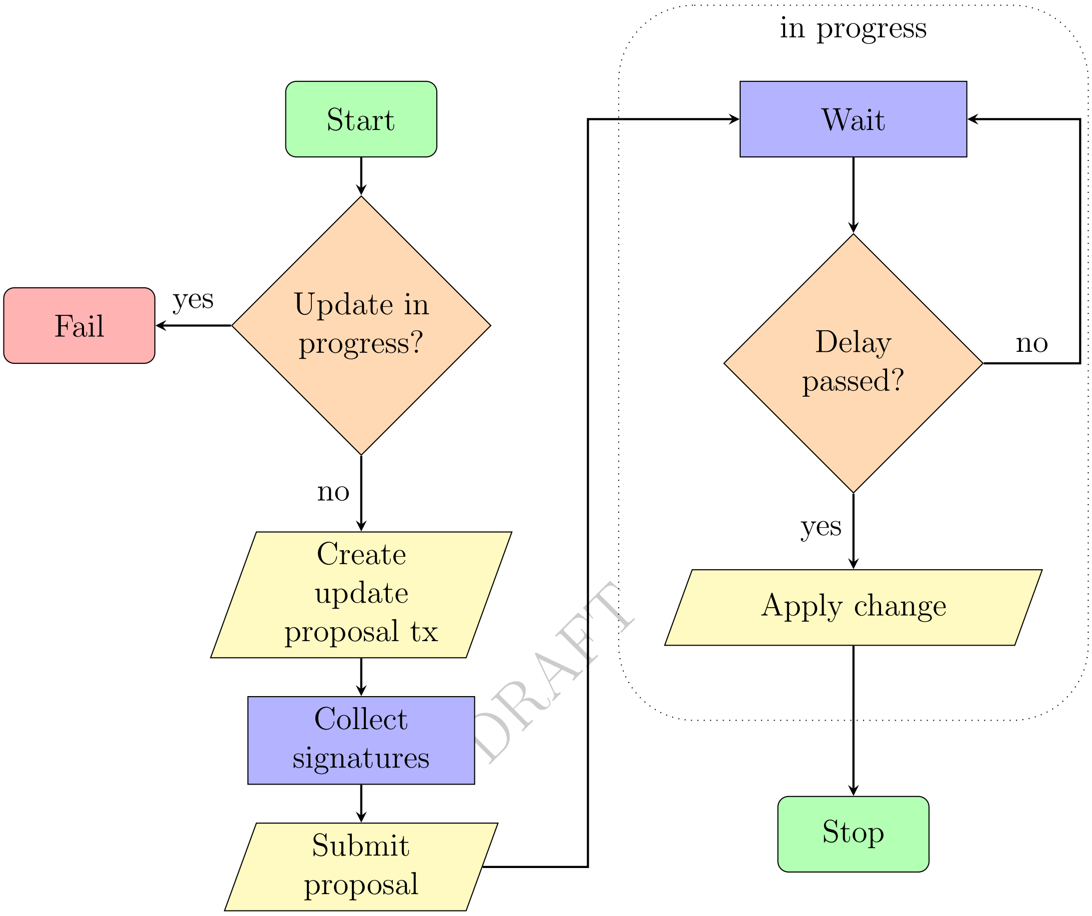


Figure 2: governance flowchart for updating DVP parameters

The voting mechanism itself, referred to as the governance delegate, is updateable. Table 4 describes 3 example governance delegates.

Complexity	**Description**
Low	A multi-signature script
Mid	Two multi-signature scripts, one for the critical parameters, and another for the non-critical parameters. The multi-signature script hashes are specified in special UTxOs which are locked at the multi-signature addresses themselves, so they can be updated using the same quorums.
High	A DAO voting hierarchy with publicly distributed DAO tokens

Table 4: example governance delegates

Parameter	UTxO	Actions	Critical?
Oracle delegate hash	`config`	Vote to change the oracle Change the oracle	**yes**
Governance delegate hash	`config`	Vote to change governance Change governance	**yes**
Update delay	`config`	Vote to change governance Change governance	**yes**
Agent public key hash	`config`	Vote to change the agent Change the agent	no
Investment universe (asset classes)	`assets`	Vote to add asset class Add asset class Vote to remove asset class Remove asset class	**yes** no

Success fee benchmark
Success fee progressive steps
Success fee period
	`config`
	
	`supply`
		Vote to update the success fee
		Update the success fee
			no

Maximum token supply	`config`	Vote to increase max token supply Increase the max token supply	no
Relative mint fee Minimum mint fee	`config`	Vote to change the mint fee Change the mint fee	no
Relative burn fee Minimum burn fee	`config`	Vote to change the burn fee Change the burn fee	no
Relative management fee Management fee period	`config`	Vote to change the management fee Change the management fee	**yes**
Maximum price age	`config`	Vote to change the max price age Change the max price age	**yes**
Metadata	`(100)`	Vote to change the metadata Change the metadata	no

Table 5: updateable DVP parameters.

8 Oracles

An oracle service is a third-party service that makes off-chain data available on-chain. Oracles are typically used for accessing asset price data with on-chain validators. Oracles are a critical component of DVPs.

All state-of-the-art eUTxO oracles are datum-based. Different oracles have different datum structures, so DVPs can't reference oracle datums directly.

To maintain flexibility, DVPs delegate oracle price feed validations to an oracle delegate, ensuring price feeds are correctly copied from oracle-specific datums into the asset counter datum price fields.

The oracle delegate can be changed through governance.

Note: oracle independence

If the DVP manager is able to influence an oracle price feed, they would be able to lower the price of an underlying asset, update the DVP token price, mint cheaply, revert the price and burn expensively, extracting all value from the DVP in one movement.

Note: ability to switch oracles

A DVP cannot rely on a single, hardcoded oracle. An oracle can go offline, or its reputation might become tainted due to malfeasance. DVPs must maintain the ability to change oracle. By changing oracles through the delayed governance process, token holders are given time to exit the DVP if they don't trust the change.

Note: price feed delays

Due to the volatility of on-chain assets and limitations of Cardano's on-chain time (accurate to about 5 minutes), significant differences can occur between oracle price feeds and market prices. This provides arbitrage opportunities for the DVP manager, which must be compensated for by lowering the fees.

9 Staking

ADA is essential for securing blockchain consensus through staking. Smart contracts can also participate in staking by using staking validators. This would however add significant complexity to the DVP smart contract.

Instead DVPs do not directly participate in staking and use so-called unstaked enterprise addresses for the DVP smart contract. If the proportion of ADA in a DVP becomes large, it should be exchanged for "staked-ADA" wrapped tokens.

Staked-ADA wrapped tokens use an independent staking validator that ensures the staking rewards are added to the reserves of those wrapped tokens. This effectively tokenizes the blockchain staking rewards.

By choosing precisely which staked-ADA wrapped tokens to hold, a DVP can participate indirectly in securing blockchain consensus.

10 Compliance

Blockchains are permissionless networks where any entity can send funds to any other entity. This is problematic as illicit funds can contaminate clean funds when mixed at a payment address.

On account-based blockchains this is difficult to avoid as an address balance is simply a number, and illicit funds entering that balance immediately contaminate that balance's clean funds. Compliant account-based smart contracts thus require special rules to limit such transfers.

On UTxO-based blockchains this is easier to avoid simply because each UTxO remains a separate entity, and can't contaminate the receiving address unless the receiver decides to spend that UTxO. So before spending UTxOs with an unknown origin, such UTxOs must be checked against chain-analysis databases before proceeding.

All DVP transactions that are at risk of contaminating the contract must be signed by the DVP manager. We refer to this signing key as the *agent*. Only order cancellation and governance updates don't require an agent signature.

10.1 Secondary markets

Secondary market trading of DVP tokens is legally beneficial. Few jurisdictions permit selling financial services without KYC and proof of investor accreditation. However, what those investors then do with those tokens is their own legal responsibility. The entity operating the DVP doesn't carry the legal burden of secondary market trading.

A Tokens and datums

Token names and on-chain data structures must be defined before defining the requirements of the validators and the topology of the transactions.

A.1 Tokens

A DVP consists of the tokens named in table 6. (100), (222) and (333) are CIP-68 prefixes. The internal stateful tokens don't use token name prefixes because the CIP-68 datum structure would create too much overhead. The voucher and asset group ids are 1-based indices. The reimbursement period id can be arbitrarily set but must increase monotonically (eg. 2024, 2025, etc.).

Name	**Description**	**Unique**
(100)	token metadata	yes
(333)	**the DVP token itself**	no
assets <group-id>	asset prices and counts	series
config	governance parameters	yes
portfolio	summary of assets	yes
price	token price	yes
reimbursement <period-id>	success fee reimbursement marker	series
supply	token and voucher supply	yes
(100)voucher <voucher-id>	voucher reference token	series
(222)voucher <voucher-id>	voucher user token	series

Table 6: overview of the tokens minted using the fund policy.

> **Note: ticker**
>
> The ticker isn't part of the token names because it wouldn't be updateable (though the metadata ticker **is** updateable).

A.2 Datums

Table 7 contains an overview of the datum structure of each stateful UTxO type.

UTxO Type	**Fields**
Mint order	Return address Return datum Minimum number of tokens Maximum price age $\tau_{order,p}$
Burn order	Return address Return datum Minimum ADA equivalent **or** minimum value Maximum price age $\tau_{order,p}$
Metadata (100)	Name Description Decimals Ticker Website URL Logo URI

Assets group
assets <id>
	List of:
	    Asset class
	    Count n_a
	    Count tick k_a
	    Price p_a
	    Price timestamp t_a

Updateable parameters
config
	Agent public key hash
	Fees:
	    Relative mint fee ϕ_m
	    Minimum mint fee $\hat{\delta}_m$
	    Relative burn fee ϕ_b
	    Minimum burn fee $\hat{\delta}_b$
	    Relative management fee ϕ_μ
	    Management fee period duration τ_μ
	    Success fee progressive steps (c_i and σ_i)
	    Success fee benchmark delegate hash
	Token:
	    Maximum token supply N_{max}
	    Maximum price age τ_p
	Oracle delegate hash
	Governance:
	    Update delay τ_{gov}
	    Delegate hash

UTxO Type	Fields

Updateable parameters (cont.)
`config`
	State as one of:
	`Idle`
	`Changing`
	Timestamp of current proposal t_{gov}
	Proposal as one of:
	`AddingAssetClass`
	Asset class
	`RemovingAssetClass`
	Asset class
	`UpdatingSuccessFee`
	New success fee period duration
	New success fee benchmark delegate hash
	New success fee progressive steps
	`IncreasingMaxTokenSupply`
	New max token supply
	`ChangingAgent`
	New agent public key hash
	`ChangingOracle`
	New oracle delegate hash
	`ChangingGovernance`
	New governance delegate hash
	New update delay
	`ChangingMintFee`
	New relative mint fee
	New minimum mint fee
	`ChangingBurnFee`
	New relative burn fee
	New minimum burn fee
	`ChangingManagementFee`
	New relative management fee
	New management fee period duration
	`ChangingMaxPriceAge`
	New maximum price age
	`ChangingMetadata`
	Hash of new metadata

UTxO Type	**Fields**

Portfolio summary
`portfolio`
	Total number of asset groups
	Reduction state as one of:
	`Idle`
	`Reducing`
	Number of asset groups iterated over
	Start tick k_p
	Reduction mode as one of:
	`TotalAssetValue`
	Total asset value V
	Oldest asset price timestamp t_p
	`Exists`
	Asset class
	Found?
	`DoesNotExist`
	Asset class

Token price `price`	Total asset value V (in lovelace, numerator) Token supply N (denominator) Oldest asset price timestamp t_p
Success fee reimbursement `reimbursement <id>`	Remaining number of vouchers Success fee start benchmark price π_{ref} Success fee end benchmark price π Success fee progressive steps (c_i and σ_i)
Token and voucher supply `supply`	Tick k Token supply N Voucher supply $N_{vouchers}$ Last voucher id Number of lovelace in vault V_{ADA} Last management fee charge timestamp t_μ Success fee: Period id Start timestamp t_α Period duration τ_α Start benchmark price π_{ref}
Voucher reference token `(100)voucher <id>`	Return address Return datum Number of DVP tokens $n_{voucher}$ Benchmark price $p_{voucher}$ Period id NFT metadata (name, description, image, url)

Table 7: overview of on-chain datum structures.

B Validators

This section lists the requirements of each DVP smart contract script:

1. Fund policy (mixed use script)
2. Mint order validator
3. Burn order validator
4. Supply validator
5. Assets validator
6. Portfolio validator
7. Price validator
8. Reimbursement validator
9. Voucher validator
10. Config validator
11. Metadata validator
12. Oracle delegate
13. Benchmark delegate
14. Governance delegate

Mathematical symbols with superscript '−' denote values **before** transaction submission, and symbols with superscript '+' denote values **after** transaction submission.

For slightly more efficient lookups, the validators can depend on each other's hashes. These hash dependencies are visualized as a graph in figure 3.

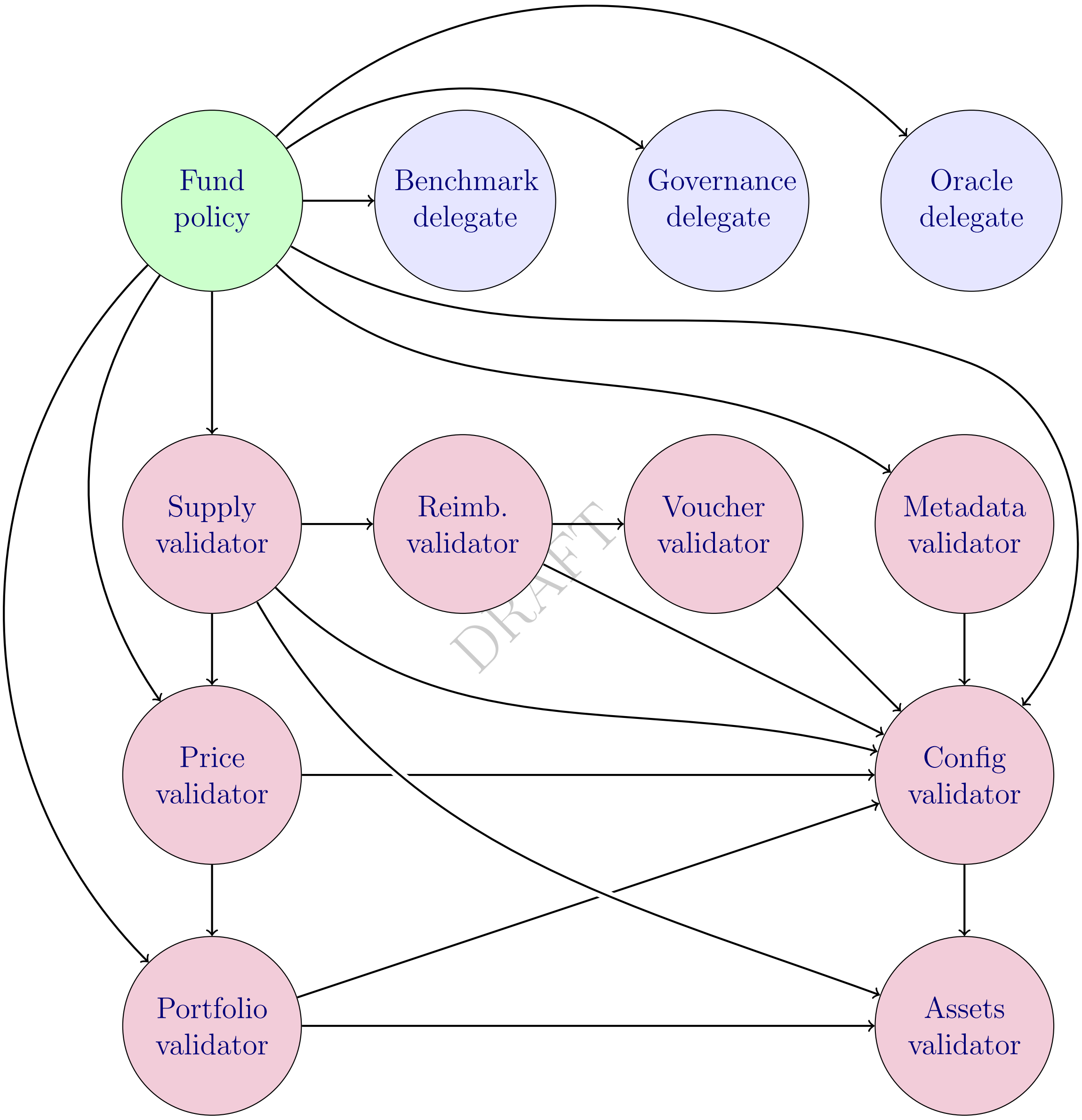


Figure 3: DAG of hash interdependencies between the DVP validators (excluding the mint and burn order validators).

B.1 Fund policy

The fund token policy, which doubles as the vault validator, witnesses the initialization of a new DVP (metadata, `config`, `portfolio`, `price` and `supply` UTxOs). For other transactions the validations are delegated to the supply validator or the portfolio validator by ensuring the `supply` or `portfolio` token is spent.

The fund policy requirements are specified in table 8.

Action / Conditions	**Requirements**
Spend Validator is called with 3 arguments	The `supply` UTxO is spent

Init
The hardcoded UTxO is spent
	Signed by the agent specified in the initial config datum
	5 tokens are minted:
	`(100)` (metadata)
	`config`
	`portfolio`
	`price`
	`supply`
	The metadata UTxO is sent to the metadata validator address, contains no other tokens, and has the expected datum
	The `config` UTxO is sent to the config validator address, contains no other tokens, and has the expected datum
	The `portfolio` UTxO is sent to the portfolio validator address, contains no other tokens, and has the expected datum
	The `price` UTxO is sent to the price validator address, contains no other tokens, and has the expected datum
	The `supply` UTxO is sent to the supply validator address, contains no other tokens, and has the expected datum
	The initial success fee is valid

Mint/burn asset group An `assets` token is minted/burned	The `portfolio` UTxO is spent
Mint other	The `supply` UTxO is spent

Table 8: requirements for the token policy and vault validator script.

B.2 Mint order validator

The mint order validator ensures a user always receives the equivalent value in return (minus fees) when the mint order is spent.

The mint order validator requirements are specified in table 9.

Action	Requirements
Cancel	Witnessed by return address spending credential
Fulfill	Signed by the agent $t_p \geq t_{tx}^{+} - \tau_p$ $t_a \geq t_{tx}^{+} - \tau_p \quad \forall a \in A_{order}$ A single return UTxO exists which matches the return address and return datum specified in the order The value difference is not larger than the allowable mint fee (calculated using the token difference) The returned number of tokens is not smaller than the minimum number of tokens specified in the order If the benchmark price is higher than the year-start price, assert that a reference voucher with the correct datum is sent to the voucher validator, and a user voucher is returned to the user

Table 9: mint order validator requirements

The fulfillment transaction fees are covered by the DVP manager, so the mint fee should be sufficient to cover these. The returned user voucher is placed in the same return UTxO to make fulfilling a bit cheaper.

The DVP manager can get away with not actually minting any tokens during this transaction and returning previously minted tokens (and vouchers) instead. From the user's perspective this doesn't matter.

The security risk of overlap between the **Cancel** and **Fulfill** actions is irrelevant because any overlap would require the transaction to be approved by the user.

B.3 Burn order validator

The burn order validator ensures a user always receives the equivalent value in return (minus fees) when the burn order is spent.

The burn order validator requirements are specified in table 10.

Action	**Requirements**
Cancel	Witnessed by return address spending credential
Fulfill	Signed by the agent $t_p \geq t_{tx}^{+} - \tau_p$ A single return UTxO exists which matches the return address and return datum specified in the order All necessary asset counters needed while calculating the value difference between the order and the return UTxO are recent The value difference is not larger than the allowable exit fee, which is calculated using the token difference and any vouchers in the order UTxO The returned value is not smaller than the minimum return value specified in the order (optionally converted to equivalent ADA)

Table 10: burn order validator requirements

Like the mint order validator, the security risk of overlap between the **Cancel** and **Fulfill** actions is irrelevant because any overlap would require the transaction to be approved by the user.

B.4 Supply validator

The supply validator is the core validator of the DVP smart contract, ensuring the security of:

- DVP token minting
- Voucher minting
- Reimbursement minting
- Asset group counting
- Vault spending (empty datum)
- `supply` UTxO spending (k, N, $N_{vouchers}$, V_{ADA}, t_μ, period id, t_α, τ_α, p_0)

To prevent inconsistencies between the action and the actual transaction, the actions are derived from the transaction context itself. Each action has the following general requirements:

The action-specific requirements are specified in table 11. The order of the action conditions is important.

Action / Conditions	DVP token minting	Voucher minting	Reimbursement minting	Asset group counting	Vault spending	**Requirements**
All						Signed by the agent The `supply` UTxO is sent back to the supply validator itself, containing only ADA and the `supply` token $k^+ = k^- + 1$ $N^+ = N^- + \Delta$ The number and position of asset groups and counters doesn't change $t^+_{tx} - t^-_{tx} < 1\text{day}$ (prevents success fee running ahead)
All except reward success						Period id doesn't change $t^+_\alpha = t^-_\alpha$ $\tau^+_\alpha = \tau^-_\alpha$ $p^+_0 = p^-_0$
All except reward management						$t^+_\mu = t^-_\mu$

Action / Conditions	DVP token minting	Voucher minting	Reimbursement minting	Asset group counting	Vault spending	Requirements
Reward success $t_{tx}^{+} > t_{\alpha}^{-} + \tau_{\alpha}$	✓	✗	✓	✗	✗	$\Delta = \Delta_{\alpha}$ (avoids voucher deadlock) $\Delta \geq 0$ $t_p \geq t_{tx}^{+} - \tau_p$ $t_{\alpha}^{+} = t_{alpha}^{-} + \tau_{\alpha}$ The period id is incremented by 1 $N_{vouchers}^{+} = 0$ The last voucher id is reset to 0 $V_{ADA}^{+} = V_{ADA}^{-}$ Only one `reimbursement` token, with the correct id, is minted Minted amount is sent to the reimbursement validator with the correct datum, and includes the `reimbursement` token The new year duration and benchmark price are set if the config UTxO is being spent The config UTxO must be spent if it is being used to update the success fee
Reward management $t_{\mu}^{+} \neq t_{\mu}^{-}$	✓	✗	✗	✗	✗	$\Delta \leq \Delta_{\mu}$ $\Delta \geq 0$ $t_{\mu}^{-} \leq t_{tx}^{-} - \tau_{\mu}$ $t_{\mu}^{+} \geq t_{tx}^{+}$ $t_{\mu}^{+} < t_{tx}^{+} + \tau_{\mu}$ (prevents unbounded datum) $N_{vouchers}^{+} = N_{vouchers}^{-}$ The last voucher id doesn't change $V_{ADA}^{+} = V_{ADA}^{-}$

Action / Conditions	DVP token minting	Voucher minting	Reimbursement minting	Asset group counting	Vault spending	**Requirements**
Mint user tokens $\Delta > 0$	✓	✓	✗	✓	✓	$t_p \geq t_{tx}^{+} - \tau_p$ $V^{+} - V^{-} \geq p \cdot (N^{+} - N^{-})$ $N^{+} \leq N_{max}$ The asset counters increase accordingly The asset prices are recent Each voucher is minted as a reference token and a user token Each voucher reference token is sent to the voucher validator address with the correct datum Each voucher has a unique id $N_{vouchers}^{+}$ increases by the number of voucher minted The last voucher id increases by the number of vouchers minted
Burn user tokens $\Delta < 0$	✓	✓	✗	✓	✓	$t_p \geq t_{tx}^{+} - \tau_p$ $V^{+} - V^{-} \geq p \cdot (N^{+} - N^{-})$ The asset counters decrease accordingly The asset prices are recent For each voucher both the reference token and the user token are burned $N_{vouchers}^{+}$ decreases accordingly The last voucher id doesn't change
Swap assets	✗	✗	✗	✓	✓	$V^{+} \geq V^{-}$ The asset prices are recent The asset counters change accordingly $N_{vouchers}^{+} = N_{vouchers}^{-}$ The last voucher id doesn't change

Table 11: supply validator action requirements.

B.5 Assets validator

The assets validator ensures the datums of the `assets` UTxOs are correctly updated. These validations are delegated to the supply validator and the portfolio validator.

We can simply require that either the supply or the portfolio validator witness the transaction. There is no issue with these two overlapping:

1. Any transaction witnessed by the supply validator only allows changes to the asset counts and ticks
2. Any transaction witnessed by the portfolio validator doesn't allow changes to the asset counts and ticks, but does allow changes to asset class positions and prices, and adding/removing asset classes

B.6 Portfolio validator

The portfolio validator ensures all the assets are correctly iterated over when determining one of the following:

- Total asset value
- Asset class exists in the investment universe
- Asset class doesn't exist in the investment universe

The portfolio validator also secures asset group minting, and asset group spending actions unrelated to counting (asset counts and ticks can never change when the transaction is witnessed by this validator).

The portfolio validator requirements are specified in table 12. There is no risk of the actions overlapping because they are derived from the transaction context.

Action / Conditions	**Asset group minting**	**Asset group spending**	**Requirements**
All			Signed by agent
All except add/remove asset group			The number of asset groups stays the same
Add asset group A single token is minted	✓	✗	`Idle` ➔ `Idle` The minted `assets` token has the correct id The number of asset groups increased by 1 The asset group is sent to the assets validator address The asset group UTxO doesn't contain any other tokens The asset group datum is an empty list
Remove asset group A single token is burned	✓	✓	`Idle` ➔ `Idle` The burned `assets` token has the correct id The number of asset groups decreased by 1 The asset group datum is the empty list

Action / Conditions	Asset group minting	Asset group spending	Requirements
Any reduction start `Idle` ➔ `Reducing`	✗	✗	$k_p^+ = k$ The number of asset groups is equal to the number of referenced asset groups The asset groups are iterated over in order
Any reduction continue `Reducing` ➔ `Reducing`	✗	✗	$k_p^+ = k_p^-$ The number of asset groups is equal to the number of referenced asset groups added to the previous value The asset groups are iterated over in order
Sum ➔ `TotalAssetValue`	✗	✗	$t_p^+ = \min(t_a) \quad \forall a \in A$ $V^+ = V_{ADA} + \sum_{a \in A} p_a \cdot n_a$ $k_a \leq k_p \quad \forall a \in A$
Continue sum `TotalAssetValue` ➔ `TotalAssetValue`	✗	✗	$t_p^+ = \min(t_p^-, \min(t_a)) \quad \forall a \in A$ $V^+ = V^- + \sum_{a \in A} p_a \cdot n_a$ $k_a \leq k_p \quad \forall a \in A$
Proving existence ➔ `Exists`	✗	✗	The found flag equals true if the asset class is in any of the referenced groups
Continue proving existence `Exists` ➔ `Exists`	✗	✗	The found flag equals true if it was previously set to true or the asset class is in any of the referenced groups The given asset class doesn't change
Proving non-existence ➔ `DoesNotExist`	✗	✗	The asset class isn't found
Continue proving non-existence `DoesNotExist` ➔ `DoesNotExist`	✗	✗	The asset class isn't found The given asset class doesn't change

Action / Conditions	Asset group minting	Asset group spending	Requirements
Add asset class	✗	✓	`DoesNotExist` ➔ `Idle` A single asset group is spent and returned to the assets validator address The **DoesNotExist** state specifies the added asset class The new asset class is appended to the group list, with the correct data The list isn't larger than the allowed max size The config UTxO is consumed, proving the given asset class was intended to be added
Remove asset class	✗	✓	`Exists` ➔ `Idle` A single asset group is spent and returned to the assets validator address The **Exists** state specifies the removed asset class The removed asset class count is 0 The asset class is removed from the group list, the other list entries remain the same The config UTxO is consumed, proving the given asset class was intended to be removed
Update assets	✗	✓	`Idle` ➔ `Idle` Witnessed by the oracle delegate Each input asset group is also found as an output The asset classes are found in the same positions in the same groups Only the price and the price timestamp can change for each asset
Move assets	✗	✓	`Idle` ➔ `Idle` Each asset class in the asset group inputs is encountered in the outputs Each asset class in the asset group outputs is encountered in the inputs The total number of asset classes in the inputs and the outputs is the same None of the asset group lists exceeds the limit size
Reset	✗	✗	`Reducing` ➔ `Idle`

Table 12: portfolio validator requirements

B.7 Price validator

The price validator ensures the `price` UTxO datum is correctly updated.

The requirements of the price validator are specified in table 13.

Action	Requirements
Update	Signed by the agent
	The referenced `portfolio` UTxO is in `TotalAssetValue` state
	N is copied from the referenced `supply` UTxO datum
	t_p is copied from the `portfolio` UTxO datum
	V is copied from the `portfolio` UTxO datum

Table 13: price validator requirements

B.8 Reimbursement validator

The reimbursement validator ensures all voucher reimbursements are correctly tracked, and that the remaining success fee can only be extract once all reference vouchers of the given period have been burned. The voucher validator delegates its reimbursement validations to this validator.

The reimbursement validator requirements are specified in table 14. The actions are derived from the transaction context.

Action / Conditions	**Requirements**
All	Signed by agent For each voucher the right amount is sent to the return address with the return datum Each voucher is burned
Extract The number of vouchers burned is equal to $N_{vouchers}$	The reimbursement token is burned
Reimburse	The number of vouchers burned is deducted from $N_{vouchers}$ The remaining tokens are sent to the same address as part of the `reimbursement` UTxO The other reimbursement datum fields remain unchanged

Table 14: reimbursement validator requirements

B.9 Voucher validator

The voucher validator ensures either the voucher user token is burned, or the right amount is reimbursed (delegated to the reimbursement validator).

The voucher validator requirements are specified in table 15. The actions are derived from the transaction context.

Action / Conditions	**Requirements**
Burn The voucher user token is burned	Signed by the agent The voucher reference token is burned
Reimburse	A `reimbursement` token is spent with the same period id

Table 15: voucher validator requirements

B.10 Config validator

The config validator ensures the governance delegate witnesses any parameter updates, and that parameter updates are applied after a delay. Parameter updates must be applied, and the correctness of each update is ensured during the voting process.

Each action has the general requirement that the config UTxO is returned to the config validator address. The config validator action-specific requirements are specified in table 16.

1. Vote to add asset class
2. Vote to remove asset class
3. Vote to update the success fee
4. Vote to increase the max token supply
5. Vote to change the agent pubkeyhash
6. Vote to change the oracle delegate
7. Vote to change governance
8. Vote to change the mint fee
9. Vote to change the burn fee
10. Vote to change the management fee
11. Vote to change max price age
12. Vote to change the metadata
13. Add asset class
14. Remove asset class
15. Update success fee
16. Update the max token supply
17. Change agent
18. Change oracle
19. Change governance
20. Change mint fee
21. Change burn fee
22. Change management fee
23. Change max price age
24. Change metadata

Action / Conditions	**Requirements**
All `Idle` ➔	Witnessed by the governance delegate All non-state fields in the `config` datum remain unchanged
All ➔ `Idle`	$t_{tx}^{-} \geq t_{gov}^{-} + \tau_{gov}$
1. Vote to add asset class `Idle` ➔ `AddingAssetClass`	The referenced state UTxO proves the non-existence of the asset class
2. Vote to remove asset class `Idle` ➔ `RemovingAssetClass`	The referenced state UTxO proves the existence of the asset class
3. Vote to update the success fee `Idle` ➔ `UpdatingSuccessFee`	The vote happens within a given time interval before the end of the year The new benchmark validator must witness the transaction The new success fee steps are valid The new success fee period is positive
4. Vote to increase max supply `Idle` ➔ `IncreasingMaxTokenSupply`	$N_{max}^{+} > N_{max}^{-}$
5. Vote to change the agent `Idle` ➔ `ChangingAgent`	Signed by the new agent
6. Vote to change the oracle `Idle` ➔ `ChangingOracle`	Witnessed by the new oracle delegate (dummy call)
7. Vote to change governance `Idle` ➔ `ChangingGovernance`	Witnessed by the new governance delegate $\tau_{gov}^{+} > 0$
8. Vote to change the mint fee `Idle` ➔ `ChangingMintFee`	${\phi_m}^{+} \geq 0$ ${\hat{\delta}_m}^{+} \geq 0$
9. Vote to change the burn fee `Idle` ➔ `ChangingBurnFee`	${\phi_b}^{+} \geq 0$ ${\hat{\delta}_b}^{+} \geq 0$
10. Vote to change management fee `Idle` ➔ `ChangingManagementFee`	${\phi_\mu}^{+} \geq 0$ $\tau_{\mu}^{+} > 0$
11. Vote to change max price age `Idle` ➔ `ChangingMaxPriceAge`	$\tau_{p}^{+} > 0$
12. Vote to the change the metadata `Idle` ➔ `ChangingMetadata`	Metadata hash is 32 bytes long

Action / Conditions	**Requirements**
13. Add asset class `AddingAssetClass` ➔ `Idle`	An `assets` UTxO is spent which doesn't contain the asset class, and its output does contain the asset class
14. Remove asset class `RemovingAssetClass` ➔ `Idle`	The associated `assets` UTxO is spent, and the output doesn't contain the asset class
15. Update success fee `UpdatingSuccessFee` ➔ `Idle`	A `reimbursement` token is minted with the correct period id
16. Increase max token supply `IncreasingMaxTokenSupply` ➔ `Idle`	
17. Change agent `ChangingAgent` ➔ `Idle`	
18. Change oracle `ChangingOracle` ➔ `Idle`	
19. Change governance `ChangingGovernance` ➔ `Idle`	
20. Change mint fee `ChangingMintFee` ➔ `Idle`	
21. Change burn fee `ChangingBurnFee` ➔ `Idle`	
22. Change management fee `ChangingManagementFee` ➔ `Idle`	
23. Change max price age `ChangingMaxTokenPriceAge` ➔ `Idle`	
24. Change metadata `ChangingMetadata` ➔ `Idle`	The metadata UTxO is spent The hash of the new metadata datum matches the parameter change

Table 16: config validator requirements.

B.11 Metadata validator

The metadata validator ensures metadata changes are specified by the config UTxO and that the metadata UTxO is returned to the same address.

The metadata validator requirements are specified in table 17.

Action	Requirements
Change	Signed by the agent The metadata UTxO is returned to the same address Config state change: `ChangingMetadata` ➔ `Idle`

Table 17: metadata validator requirements

B.12 Oracle delegate

The oracle delegate ensures prices and timestamps of any asset classes in spent `assets` UTxOs are correctly updated.

The current Cardano oracle solutions are seriously lacking in functionality, reliability, speed, cost, so this is currently just a multi-sig script.

B.13 Benchmark delegate

The benchmark delegate ensures the redeemer price ratio is equal to the oracle price feed. Initially ADA itself can be used a benchmark, which means the ratio value given as a redeemer must be equal to unity.

B.14 Governance delegate

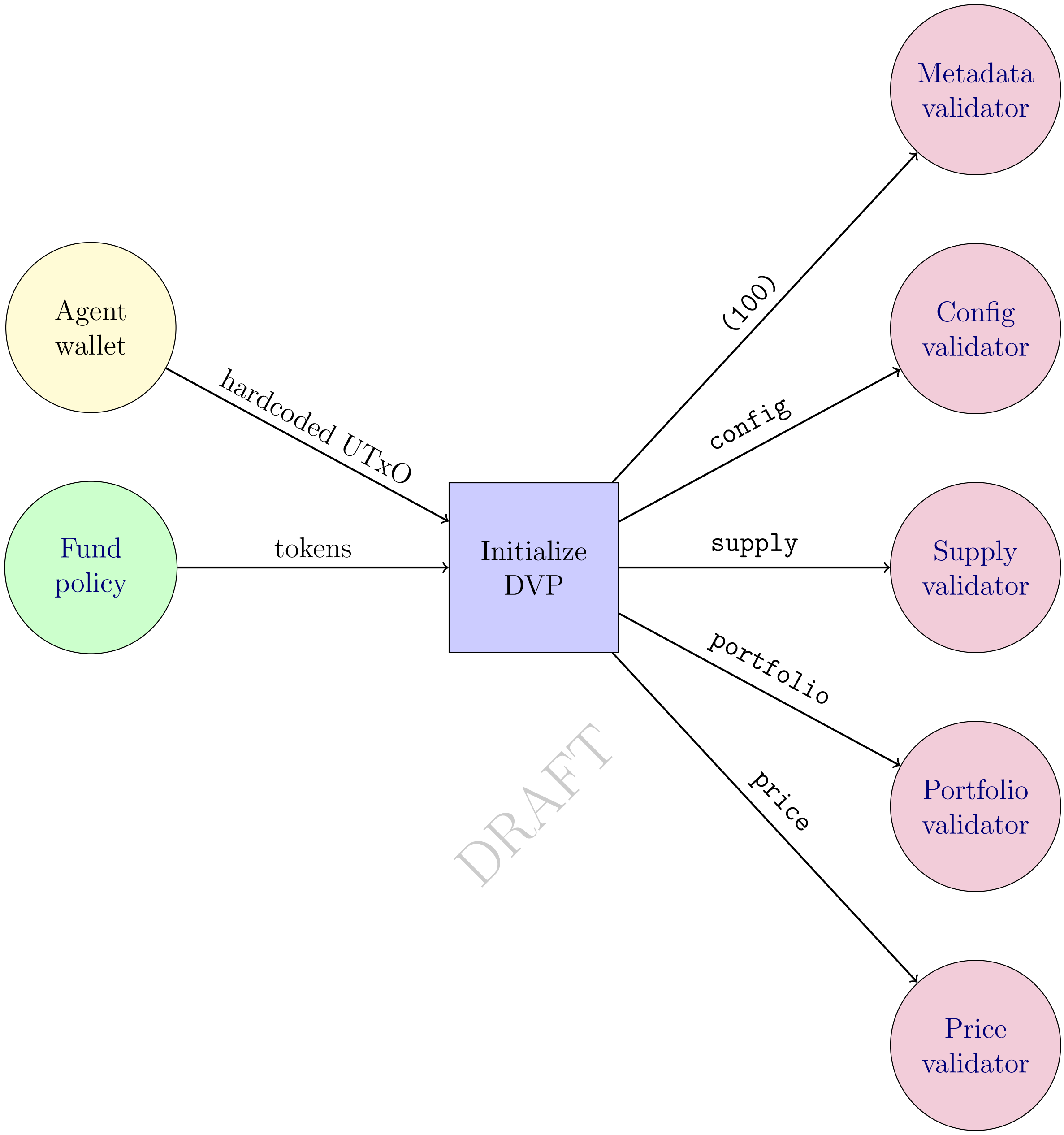
Initially, this will be a simple multi-signature script. Later, it can be upgraded to a validator wrapping two multi-signature scripts (one for critical parameters, another for non-criticial parameters), each independently updatable using its own quorum.

C Transactions

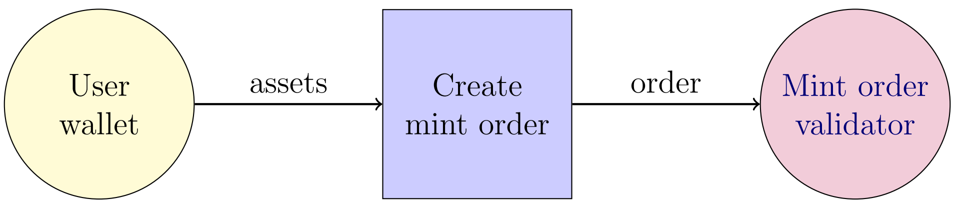
This section gives a visual overview of each DVP transaction. The network transaction fee and collateral UTxOs are omitted.

1. Initialize DVP
2. Create mint order
3. Cancel mint order
4. Fulfill mint order
5. Create burn order
6. Cancel burn order
7. Fulfill burn order
8. Swap assets
9. Update asset price
10. Count total asset value
11. Update token price
12. Add assets group
13. Prove non-existence of asset class
14. Add asset class
15. Prove existence of asset class
16. Remove asset class
17. Remove assets group
18. Move asset class
19. Reward management
20. Reward success
21. Reimburse success fee
22. Extract success fee
23. Prepare parameter update
24. Update config
25. Change metadata

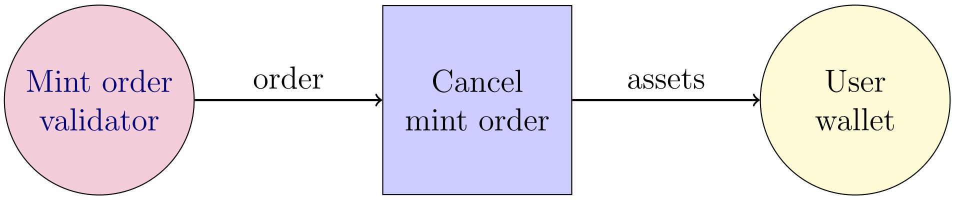
C.1 Initialize DVP



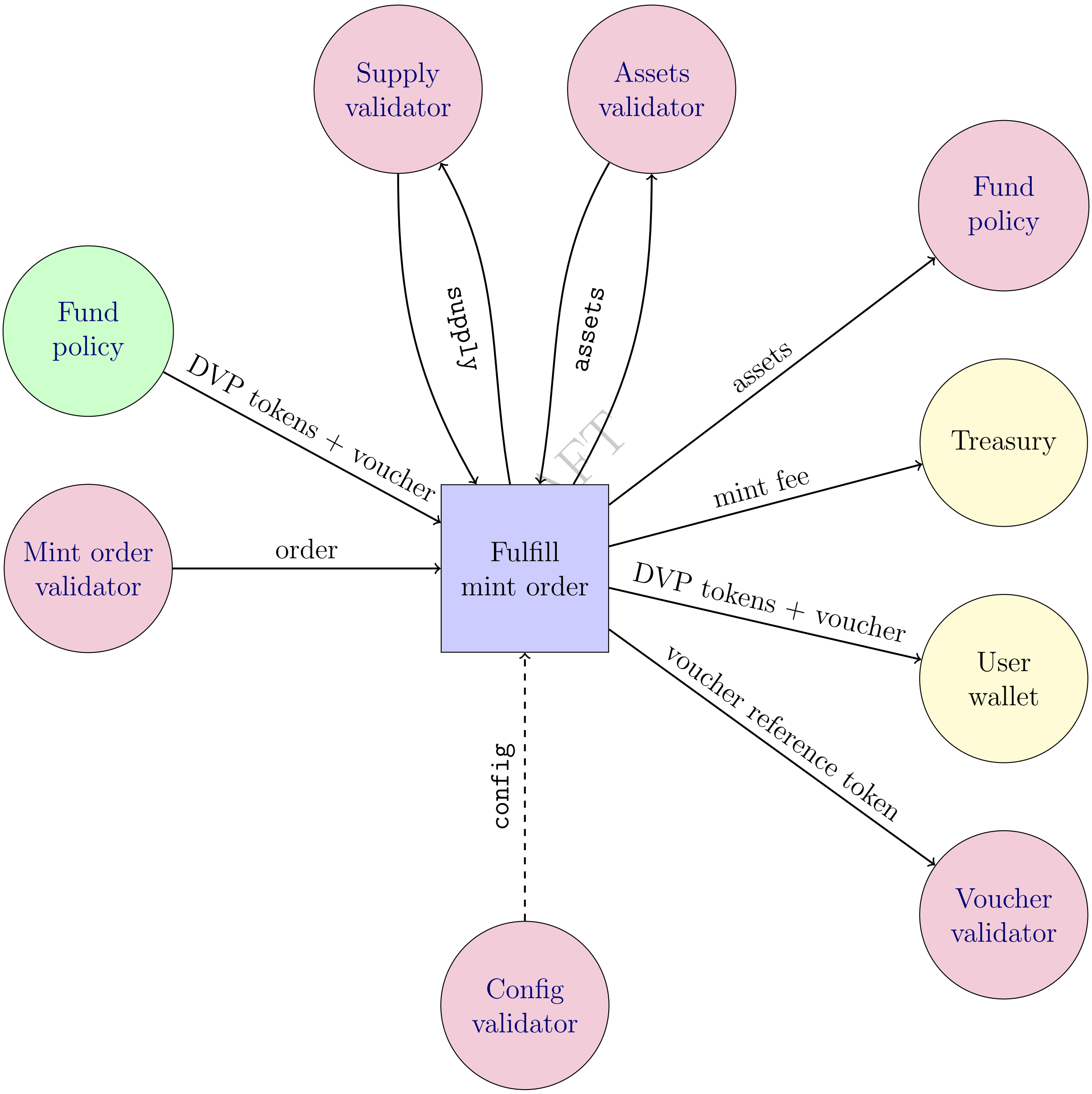
C.2 Create mint order



C.3 Cancel mint order

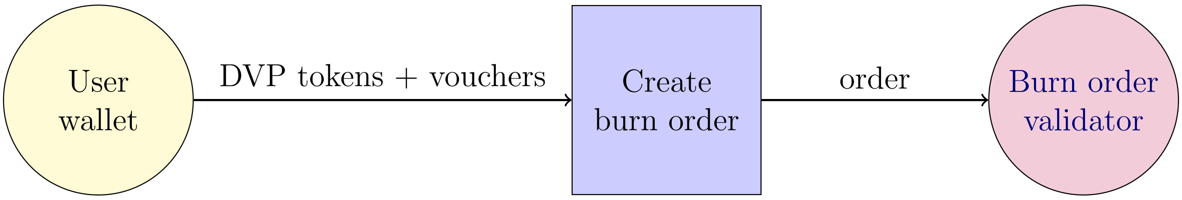


C.4 Fulfill mint order

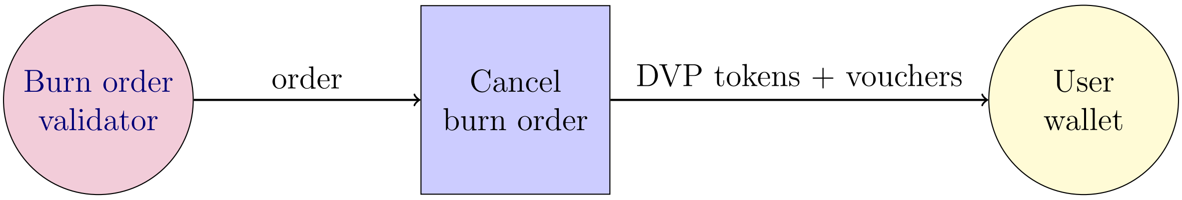


This transaction is also witnessed by the benchmark delegate.

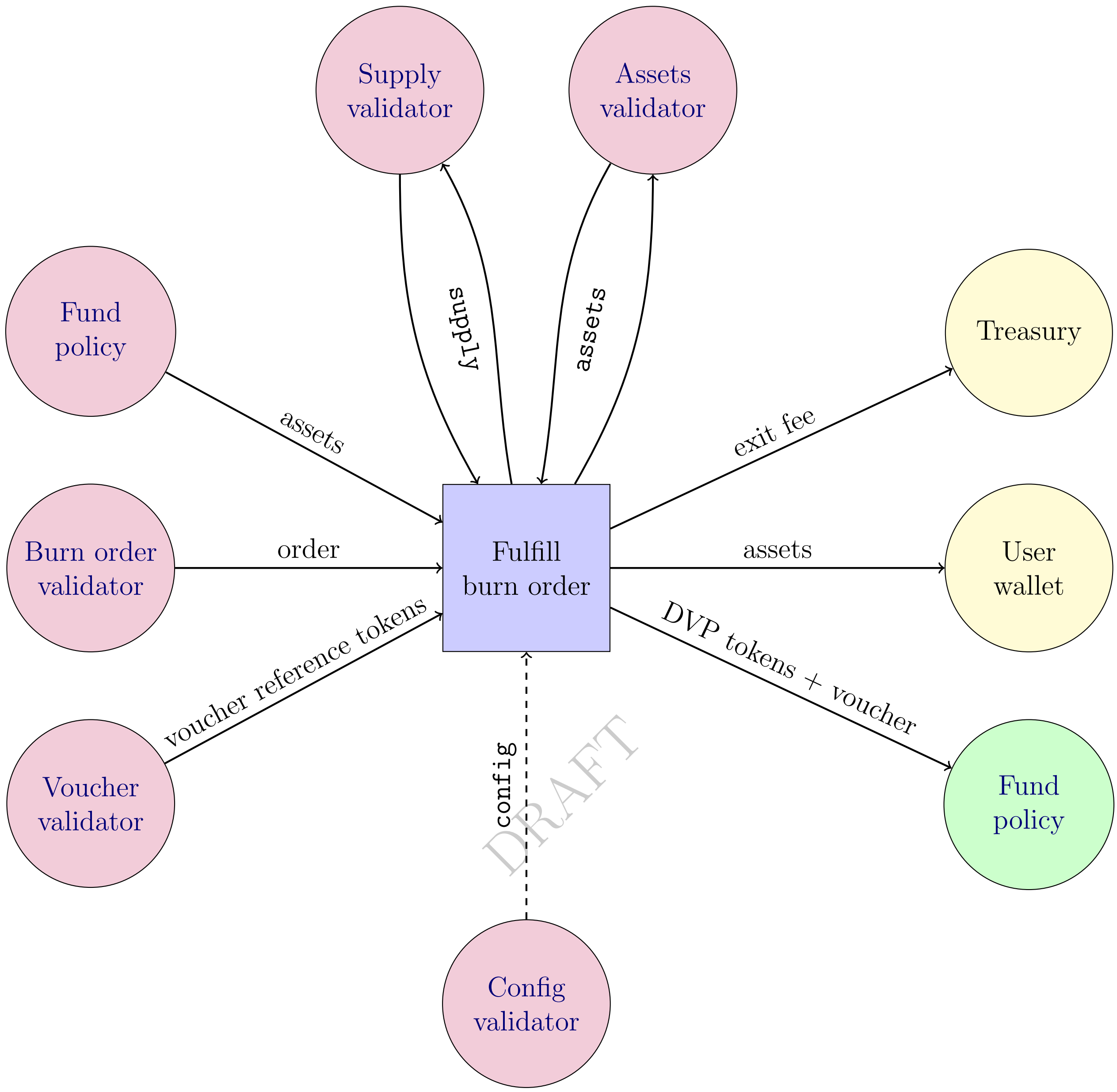
C.5 Create burn order



C.6 Cancel burn order

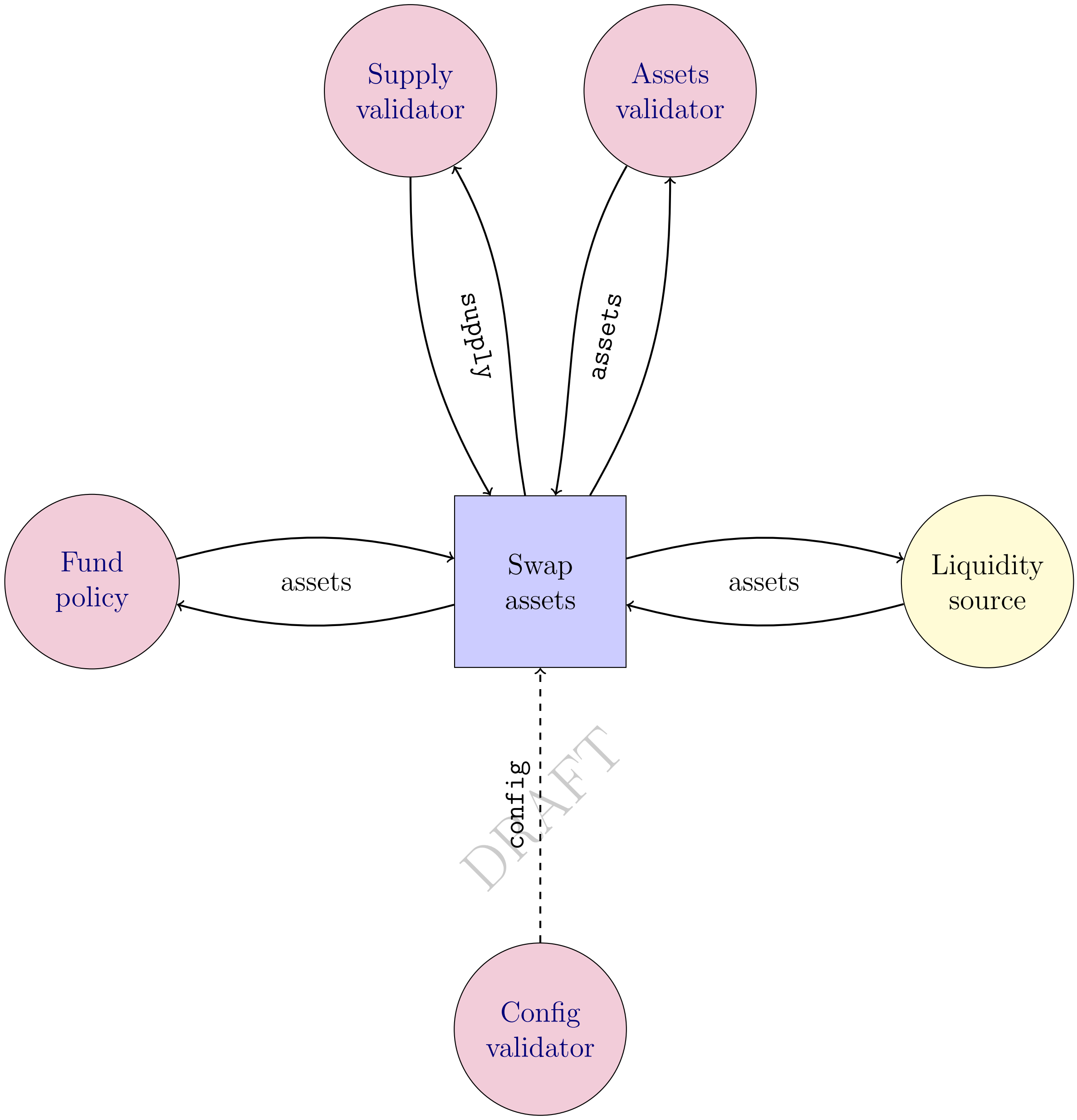


C.7 Fulfill burn order

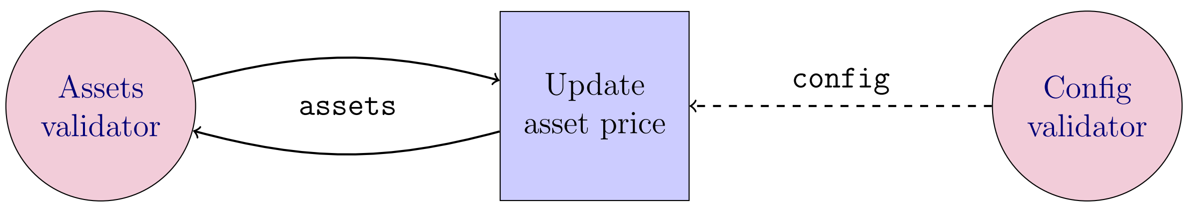


This transaction is also witnessed by the benchmark delegate.

C.8 Swap assets

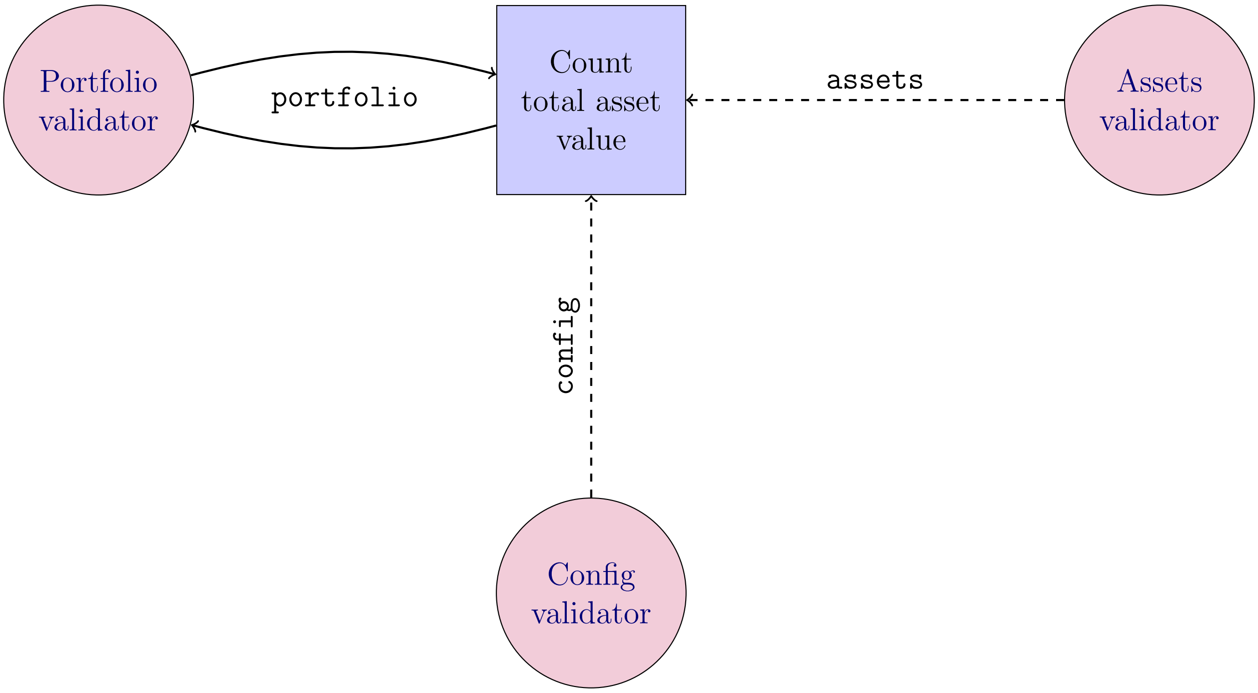


C.9 Update asset price



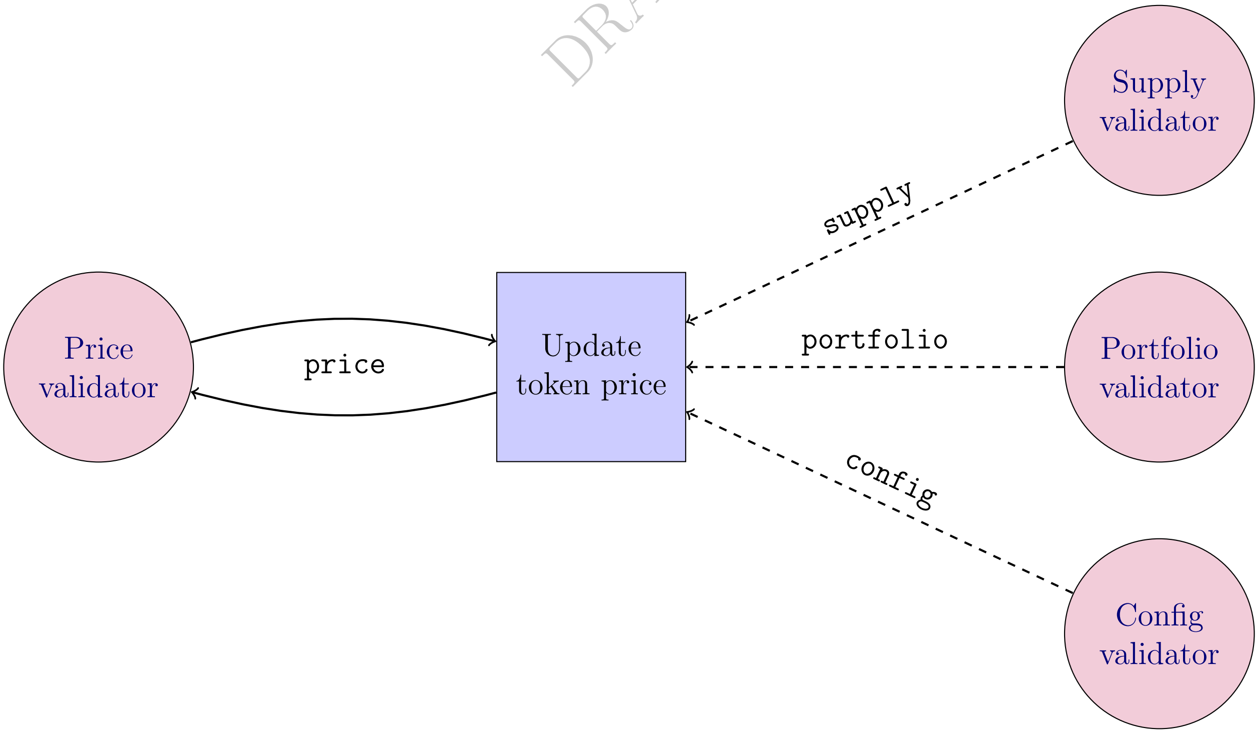
This transaction is also witnessed by the oracle delegate.

C.10 Count total asset value

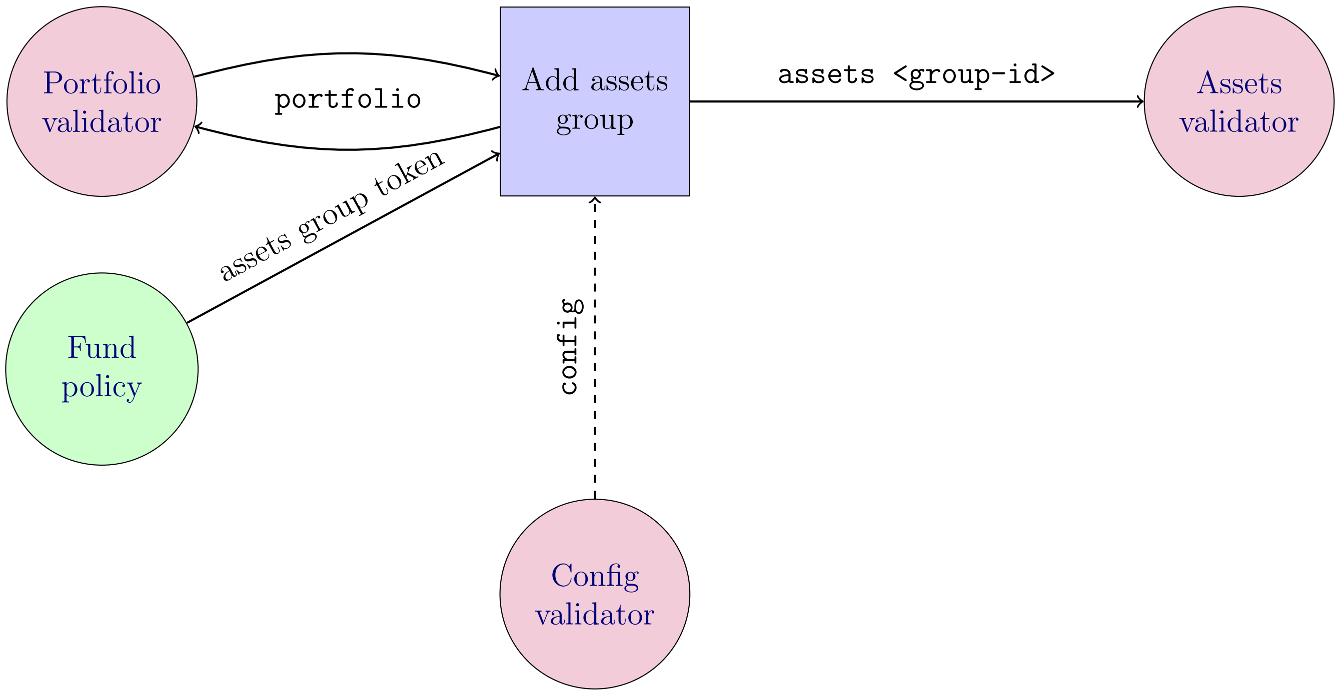


This transaction must be repeated until all asset groups are iterated over.

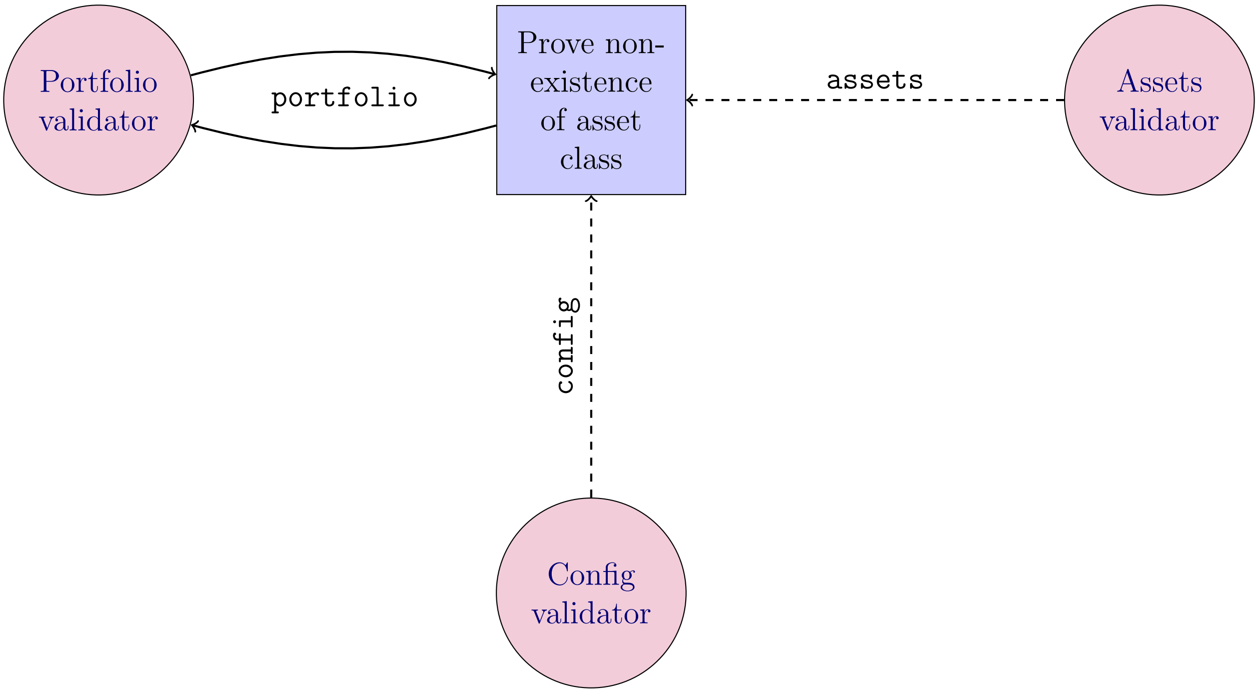
C.11 Update token price



C.12 Add assets group

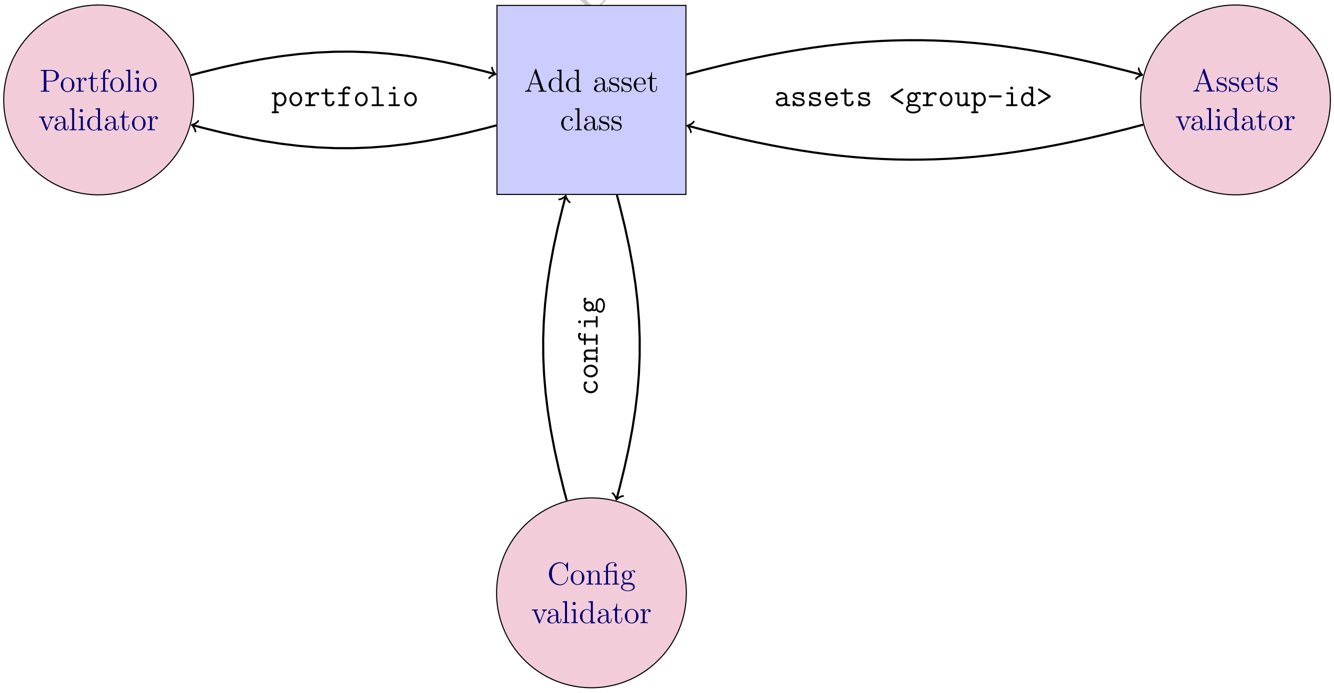


C.13 Prove non-existence of asset class



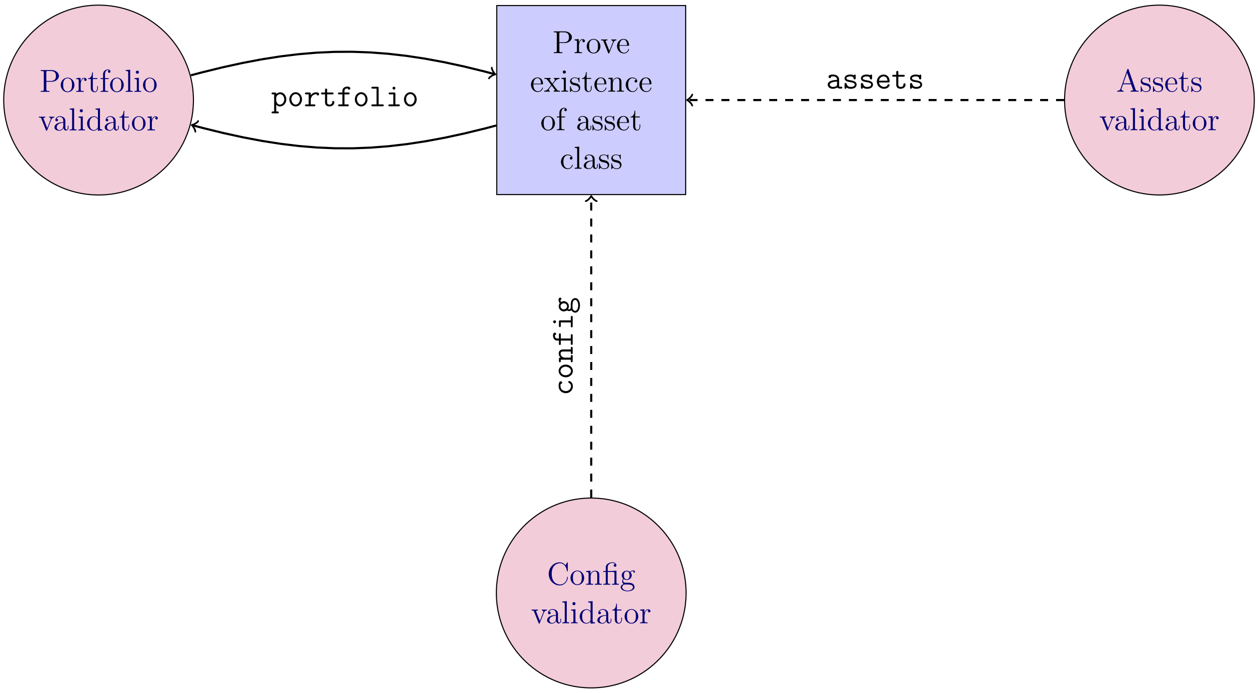
The topology of this transaction is identical to the count total asset value transaction. This transaction must be repeated until all the asset groups have been iterated over.

C.14 Add asset class



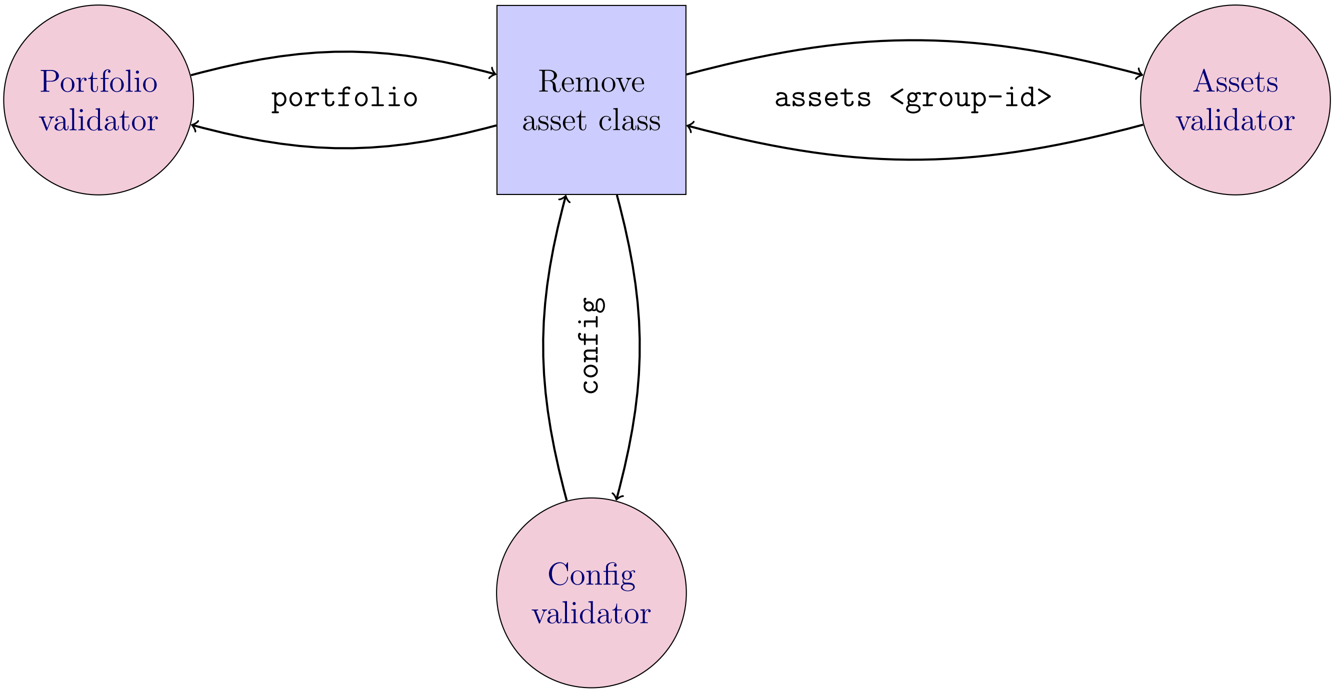
This transaction must be preceded by a proof of non-existence of the asset class.

C.15 Prove existence of asset class



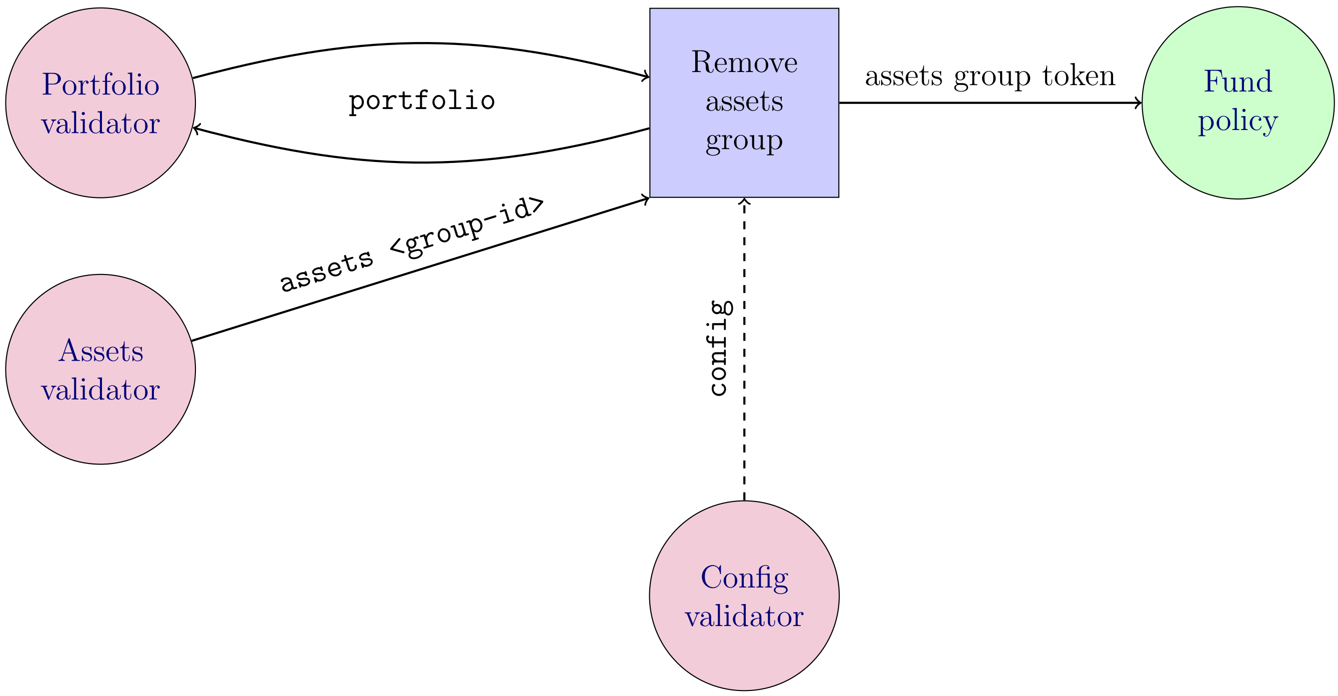
The topology of this transaction is identical to the count total asset value transaction and the prove non-existence of asset class transaction. This transaction must be repeated until the given asset class is encountered.

C.16 Remove asset class

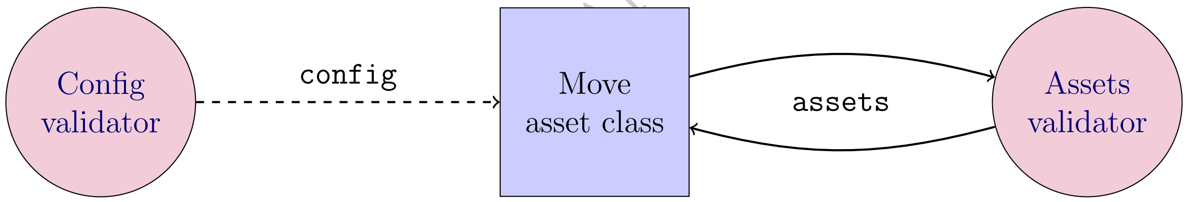


The topology of this transaction is identical to the add asset class transaction. This transaction must be preceded by a proof of existence of the asset class.

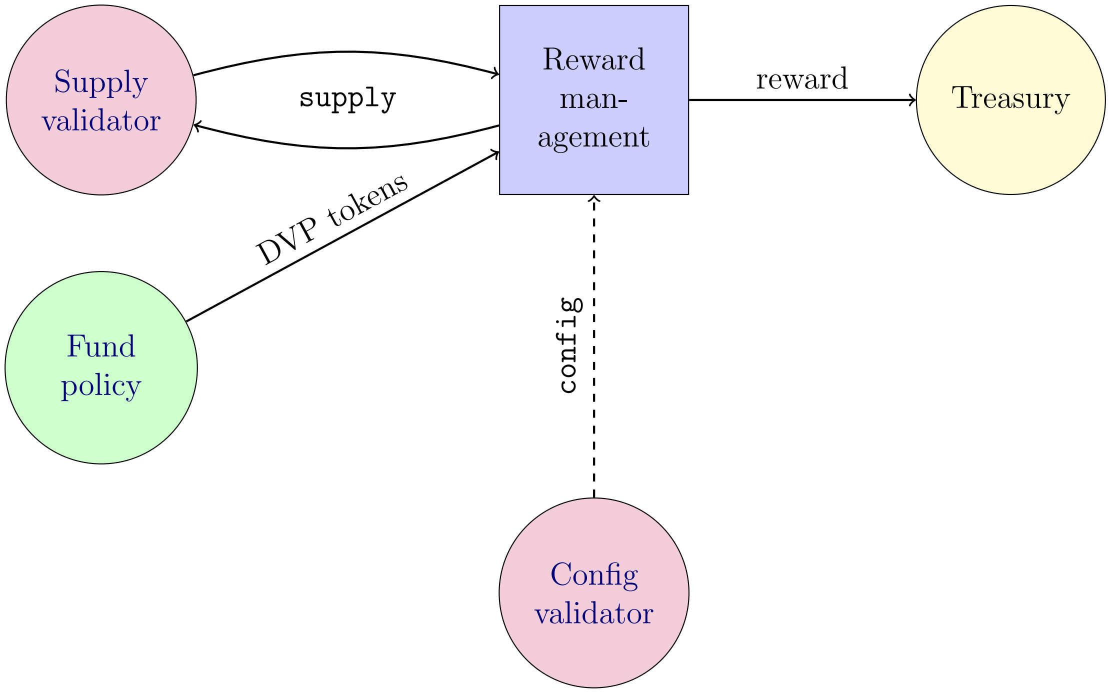
C.17 Remove assets group



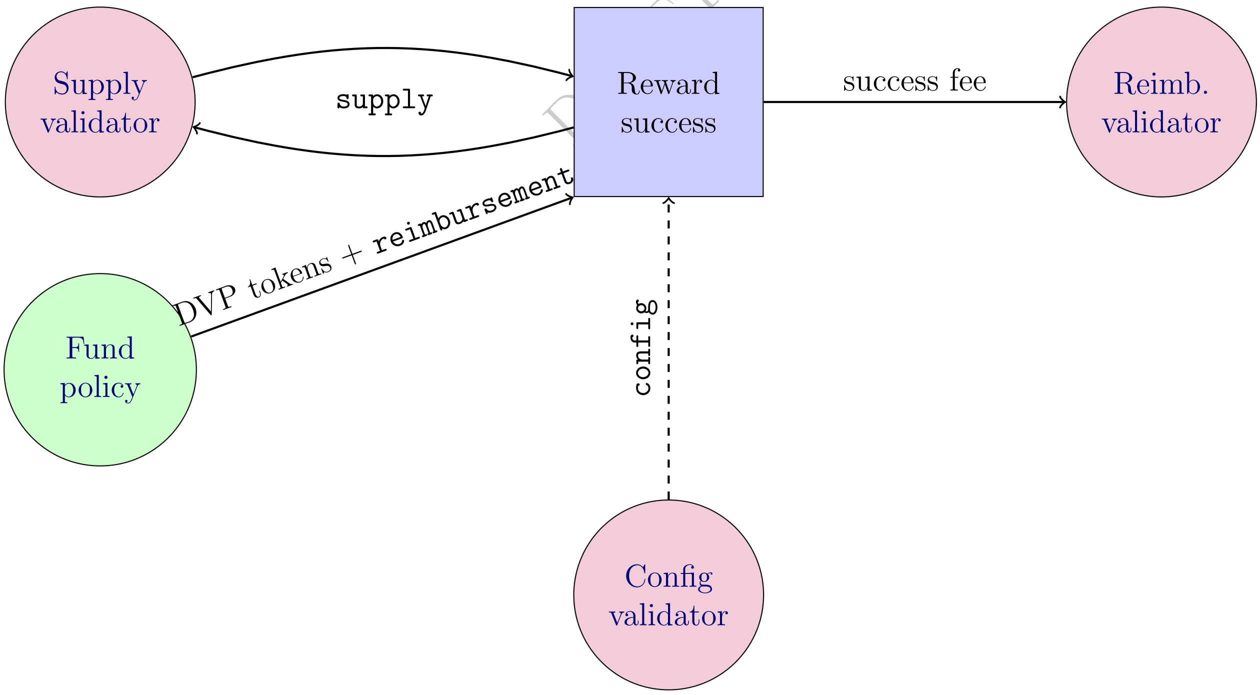
C.18 Move asset class



C.19 Reward management

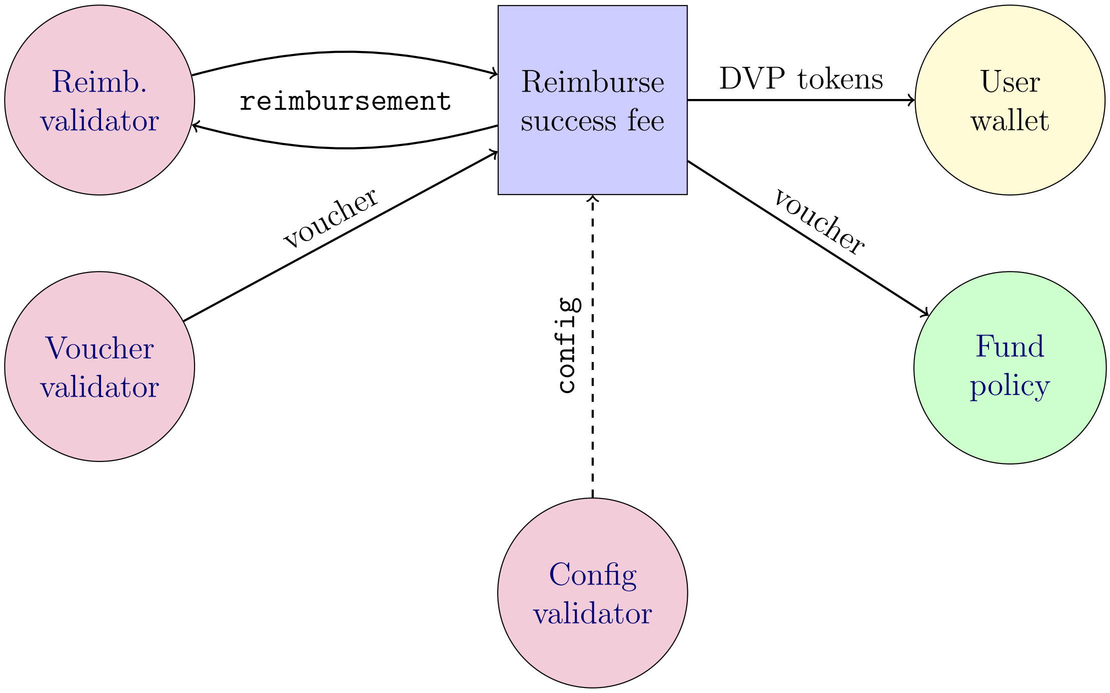


C.20 Reward success



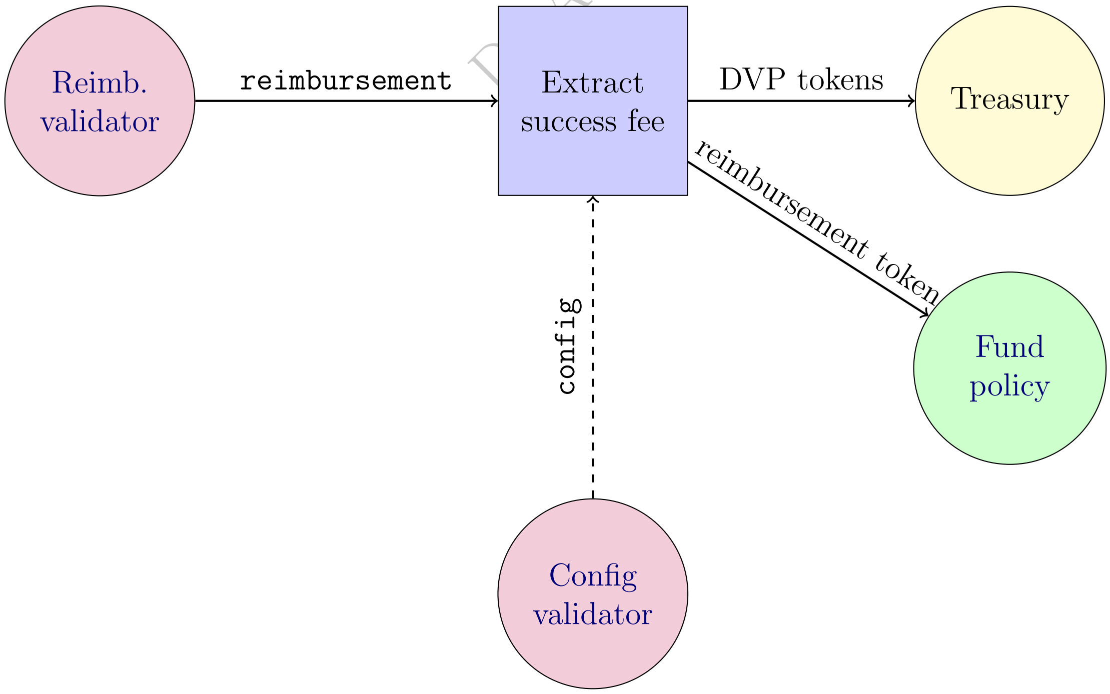
This transaction is also witnessed by the benchmark delegate.

C.21 Reimburse success fee

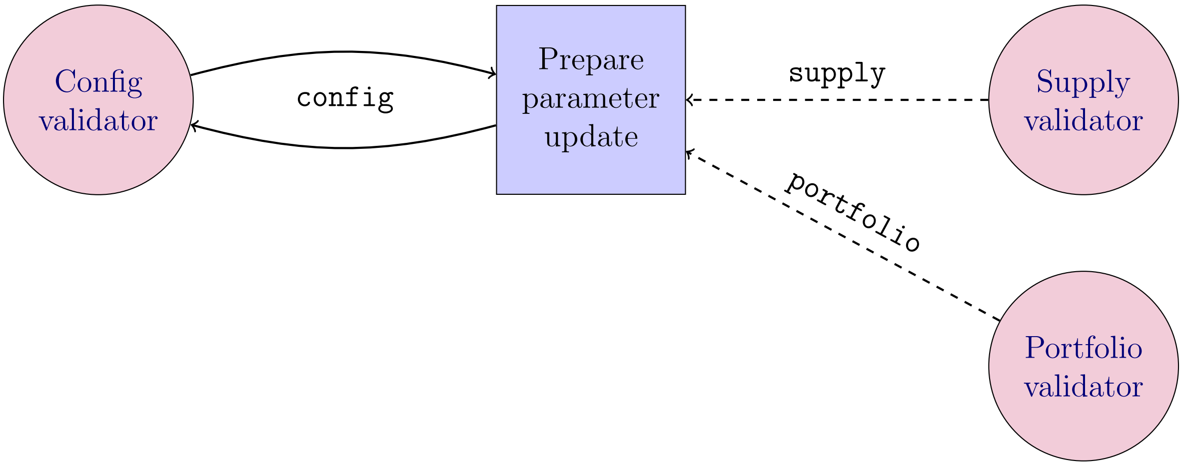


Remaining success fee is kept in the state UTxO.

C.22 Extract success fee



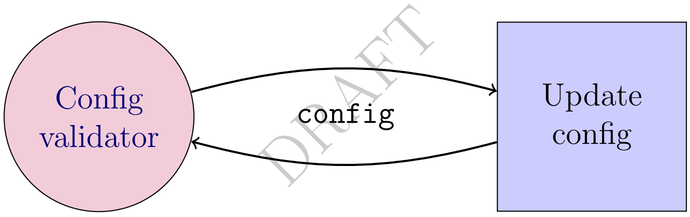
C.23 Prepare parameter update



This transaction is also witnessed by the governance delegate. This transaction is also used to prepare updates of the `config`, `assets` and `(100)` (metadata) UTxOs.

Referencing the `supply` or `portfolio` UTxOs allows checking if the update is valid (e.g. max token supply can only increase, proof of non-existence of asset class).

C.24 Update config



This transaction has the same topology as the prepare config update transaction, except that this transaction isn't witnessed by the governance delegate.

C.25 Change metadata