The Ocean

Shell Protocol v2 White Paper, Part 2

February 9, 2022

Abstract

One of the most important attributes of DeFi is the ability to compose relatively simple financial primitives to create complex financial programs. This is similar to how simple lego bricks can be assembled to construct a great variety of structures. However, composing primitives in DeFi is currently a difficult task. Most of the time, end users must send multiple transactions, waiting idly for one transaction to get mined before starting the next one. This wastes both time and money (gas). For certain use cases, special-purpose smart contracts exist to compose primitives (for example, DEX aggregators), but writing a new smart contract for every permutation of primitive interactions is infeasible. Composability in DeFi is deficient because financial primitives are implemented as monolithic protocols while tokens are siloed into separate smart contracts. What we need is an architecture where all tokens exist on a single smart contract and primitives are modular programs that share the same accounting logic. We call this system the Ocean. When composing primitives via the Ocean, no additional smart contracts are necessary and the marginal gas cost per interaction can be up to four times lower. Any type of primitive can be built on the Ocean: AMMs, lending pools, algorithmic stablecoins, NFT markets, and even primitives yet to be invented.

1 Introduction

The purpose of this paper is to reimagine decentralized finance (DeFi) architecture. At a system level, DeFi should be designed to seamlessly compose financial primitives. Examples of financial primitives include automated market makers (AMMs)[3][7], lending pools[2][14], algorithmic stablecoins[16] and non-fungible token (NFT) markets[18], among others. A simple example of primitive composition is the process of splitting a swap through multiple AMMs to get the best rate.[13] Another example is borrowing tokens on a lending pool and swapping those tokens through an AMM in order to gain leverage.[22]

In these examples, we are able to take relatively simple primitives and use them as building blocks to create complex financial programs. This attribute of DeFi is where the term “money legos”[23] gets its meaning. Lego bricks are simple shapes that can be assembled to create complex structures. In DeFi, composability opens the door to financial innovation. High composability means components are simpler to modify (notably without disruption to the larger system) and can be assembled in creative and unexpected ways.

Despite its potential, composing primitives in DeFi today is an onerous task. In most cases, a user will have to interact with one primitive at a time, sending multiple transactions, waiting idly for each transaction to clear on the blockchain. Not only is this a bad user experience (UX), it also adds gas costs and precludes use cases where transactions must execute atomically. For example, we may not want to execute a split swap unless guaranteed both trades will clear.

1.1 Sequencer-Adapter Model

Superficially, composing primitives is difficult because an off-chain user can only invoke one smart contract per transaction on the Ethereum Virtual Machine (EVM)[9]. The apparent solution to this problem is to create a special-purpose smart contract which interacts with financial primitives on the user's behalf. Smart contracts, unlike users, have the ability to interact with multiple smart contracts in a single transaction. We refer to this solution as the "sequencer-adapter" model.

The sequencer smart contract coordinates and executes the interactions, while the adapter contracts interface with the primitives (Figure 1). Decentralized exchange (DEX) aggregators such as 1inch[1] and Paraswap[19] are examples of sequencer-adapter platforms. Aggregators use off-chain algorithms to find the optimal swap split through the myriad AMMs available. They then use their sequencer-adapter contracts to atomically execute multiple swaps through multiple AMMs. InstaDapp[12] and Zapper[26] are another such example. These platforms compose lending pools and AMMs to offer services such as leveraged exposure to popular tokens.

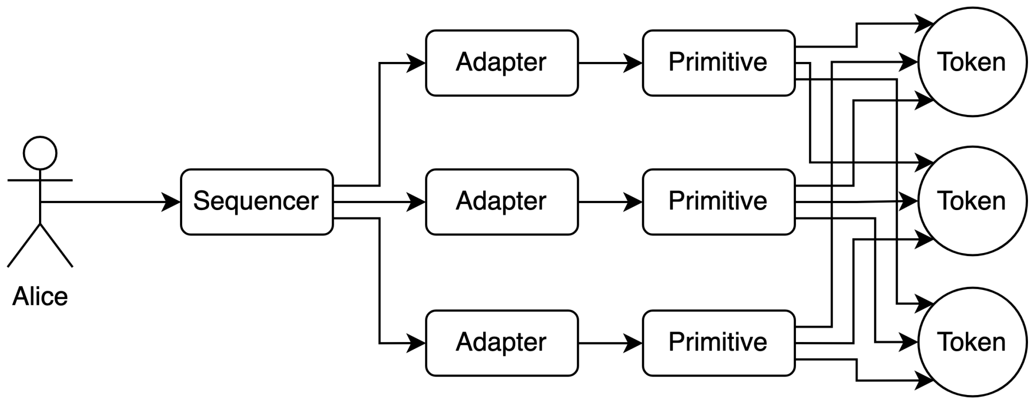


Figure 1: Simplified diagram of sequencer-adapter architecture

With the sequencer-adapter model, what previously took multiple transactions can now execute in a single, atomic transaction. This approach improves UX and can save moderate amounts of gas. The problem, however, is that every feature of every protocol requires a custom adapter. The more primitives we want to interact with, the more smart contracts we need to write and deploy. By analogy, consider a cable adapter kit that can interface with HDMI, USB, Lightning, etc. As more standards are added, our adapter kit grows to an unwieldy size. It's better than no connection, but suboptimal compared to a universal cable standard such as USB-C.

1.2 The Deeper Problem

The sequencer-adapter architecture only addresses the superficial problem of users being unable to interact with multiple smart contracts. Below this lies an even deeper issue involving the system architecture of DeFi. The fundamental composability problem stems from two interrelated sources:

- Each token exists on a unique smart contract
- Primitives are monolithic (versus modular) programs

To understand the problem with token ledgers, consider the popular fungible token standard, the ERC-20[24], where there is a separate smart contract for each token. This design works fine for one-off, peer-to-peer transfers, which was its original use case. However, composing primitives requires invoking multiple transfers between the user and their target primitives.

In the EVM, every time we invoke a new smart contract during a transaction, we must pay a fixed cost of 2,600 gas to load that contract's bytecode[6]. On top of that, any token transfer requires writing data to storage at least twice. A storage write is one of the most expensive operations in the EVM, costing between 5,000 gas to 20,000 gas[10]. The end result is needless redundancy leading to excessive costs when composing primitives that will transfer multiple ERC-20 tokens.

To understand why monolithic primitives are a problem, consider that a financial primitive can be deconstructed into two parts:

- Accounting logic: moving tokens between accounts
- Business logic: computing how many tokens need to be moved

For example, an AMM has to first calculate an exchange rate (business logic) and then transfer tokens between itself and the user (accounting logic). A lending pool has to calculate the interest rate for borrowers and lenders (business logic) and then transfer the tokens to the user (accounting logic).

Currently, primitives are designed as monolithic programs, grouping both of the above concerns into a single protocol. Separation of concerns exists purely as an internal abstraction, if at all. This kind of architecture is antithetical to the modular nature of composability. If anything, the sequencer-adapter model exacerbates system complexity. The combination of siloed token ledgers and monolithic primitives makes DeFi difficult to reason about, expensive to use, and very hard to compose.

1.3 Vault-Router Model

The "vault-router" model for AMMs implemented by Balancer v2[17] and Uniswap v3[4] is an overall improvement over the sequencer-adapter model. In this architecture, all tokens in the protocol are held by a vault contract, which serves as a unified accounting system for the AMMs. AMMs track their balances on an internal ledger maintained by the vault. Users send their swaps to the router contract, which sequences the execution. If a swap involves routing a trade through multiple pools, then the inter-pool exchanges are settled on an internal ledger. The vault then batches the final ERC-20 transfers, minimizing redundant smart contract interactions. Figure 2 shows a simplified diagram of the vault-router architecture. In this way, a user can split swaps through multiple AMMs, ensuring atomic execution and saving significant amounts of gas.

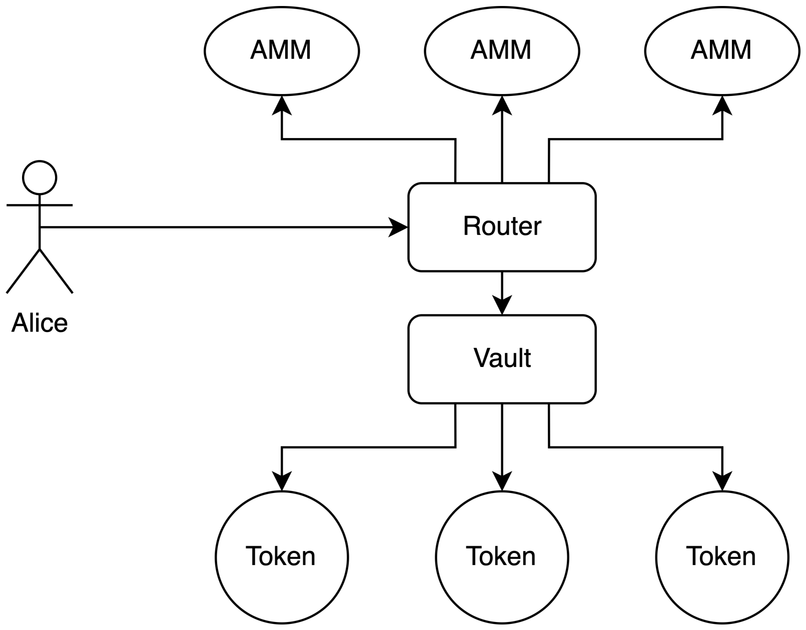


Figure 2: Simplified diagram of vault-router architecture for AMMs

In addition to gas savings, the vault-router architecture theoretically simplifies the overall system. The AMMs that integrate with the vault-router do not need to implement their own accounting logic, so the AMM developers can focus purely on business logic. Furthermore, composability is inherent to the system. No additional smart contracts are necessary beyond the core protocol to chain together a sequence of swaps.

However, current vault-router implementations are insufficient. Both Balancer and Uniswap seem to contain incidental code complexity that is unnecessary to solve the underlying problem. For example, there is no fundamental reason why the vault and the router contracts cannot be combined into one contract. In trying to streamline composability, these projects built a system that is arguably more complex than the one they wanted to replace.

Moreover, both of these implementations can only compose AMMs. In the case of Uniswap, the protocol can only handle Uniswap-controlled AMMs, creating a walled garden. At least the sequencer-adapter model, despite its shortcomings, is flexible enough to interface with any financial primitive, not just protocol-specific AMMs.

What we need is a solution that can encompass the best of both worlds: one which has the gas savings of vault-router architecture while maintaining the flexibility of sequencer-adapter architecture. And crucially, it must simplify the overall system. We call this solution the Ocean.

2 The Ocean

The objective of the Ocean is to create a platform that can compose any type of primitive: AMMs, lending pools, algorithmic stablecoins, NFT markets, and even future primitives yet to be invented. It should also support popular token standards: ERC-20, ERC-721[8] and ERC-1155[20].

2.1 Performance Criteria

The criteria to assess the Ocean's efficacy compared to other methods are:

- How much gas do we save when composing primitives on the Ocean?
- How much extra smart contract code is required to compose primitives?

Although not a formal criterion, a third consideration which had substantial influence on the design process is simplicity: How much simpler is the overall system compared to the status quo? As discussed in Section 1.2, the root of the composability problem stems from a redundant system architecture. We do not want to replace a flawed system with an even more complicated design.

2.2 Ocean Architecture

In order to get the efficiency of vault-router combined with the flexibility of sequencer-adapter, the Ocean must do the following:

- Put all tokens into a single smart contract ledger
- Generalize accounting logic to support any primitive
- When composing primitives, track intermediate balances in memory rather than storage to save gas

Figure 3 provides an overview of the Ocean's design. The "vault" equivalent is an ERC-1155 ledger that can wrap and unwrap external tokens (see Section 3). The "router" equivalent is part of the same smart contract that handles the atomic execution of Ocean-native primitive interactions (see Section 5). Ocean-native primitives implement the Ocean's accounting framework using the Ocean's ERC-1155 ledger (see Section 4). Our user, Alice, can directly interact

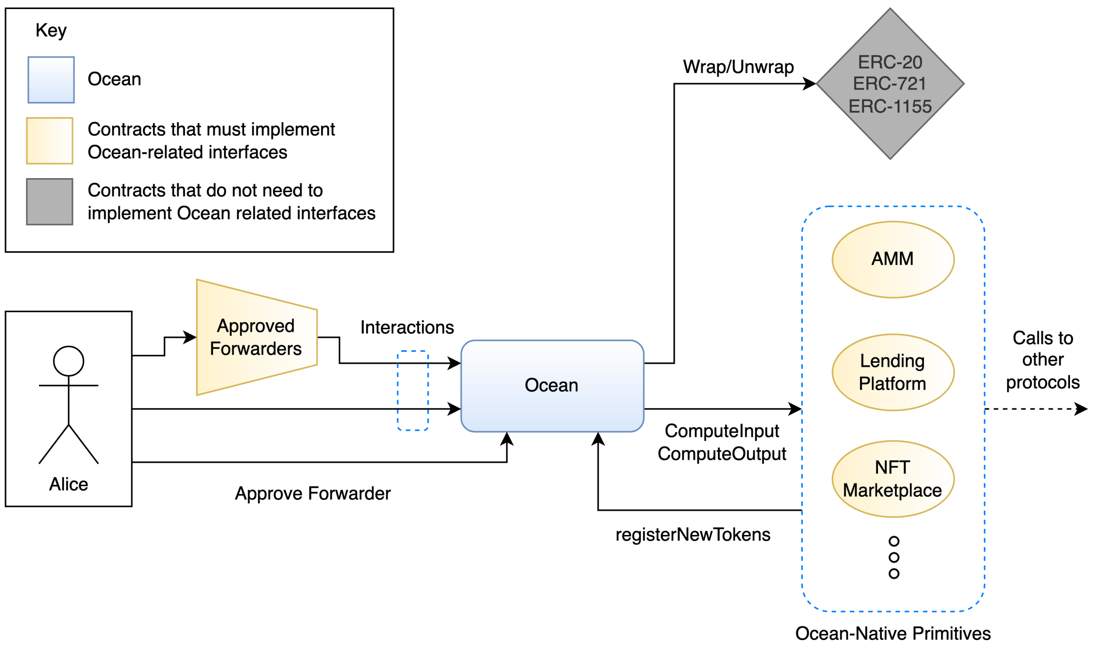


Figure 3: Complete diagram of Ocean architecture

with the Ocean, or she can designate a trusted forwarding contract to interface with the Ocean on her behalf. Forwarding contracts allow Alice to compose the Ocean with external protocols, or even compose the Ocean with itself.

2.3 Outline of the Following Sections

The following sections will explain how the Ocean works. They are intended for an audience with moderate familiarity with Solidity and the EVM. They are meant to be a companion piece to the source code, providing the intuition behind the implementation. Conceptually, we can divide our explanation into three main parts:

- Multi-token ledger
- Accounting framework for Ocean-native primitives
- Composing primitives

There is also a section to walk the reader through a simple example. Further examples are available in Appendix B.

3 Multi-Token Ledger

We chose to implement the Ocean using an ERC-1155 ledger because a single ERC-1155 can be used to represent tokens from all ERC-20, ERC-721, and ERC-1155 ledgers without any loss of information. This is due to a nested mapping architecture that tracks token balances by mapping a `uint256` (token ID) to an `address` (user's account) to another `uint256` (user's balance). For every token ID represented in the ledger, this mapping can track the balances for all users. To avoid confusion, we will refer to token IDs on the Ocean's ledger as "Ocean IDs." This naming convention distinguishes token IDs on the Ocean from token IDs on external ERC-721 and ERC-1155 ledgers.

There are two types of tokens in the Ocean's ledger:

- Tokens from external ledgers, "wrapped tokens"
- Tokens created by Ocean-native primitives, "native tokens"

3.1 Wrapped Tokens

Wrapped tokens are tokens held by the Ocean on an external ledger, with a proportional quantity represented in the Ocean ledger. We use the prefix "sh-" to denote Shell-wrapped tokens. For example, shUSDC is USDC wrapped into the Ocean. Ocean IDs for wrapped tokens are derived from their attributes on the external ledger. For ERC-20 tokens, the Ocean ID is a cast of the ERC-20 contract's address to a `uint256`:

```
uint256 oceanID = uint256(uint160(contractAddress));
```

For ERC-721 and ERC-1155 tokens, there are potentially multiple tokens represented in each smart contract. Therefore, we cannot use the contract's address as the Ocean ID. We also cannot use the token ID because that could result in a collision between two Ocean IDs. Instead, the Ocean ID is derived from a hash of the contract's address and the token ID:

```
uint256 oceanID = uint256(keccak256(abi.encodePacked(contractAddress, tokenID)));
```

3.2 Native Tokens

In contrast, a native token is a token issued by an Ocean-native primitive. An example of a native token would be LP tokens issued by an Ocean-native AMM, such as Proteus[21]. The source of truth for wrapped tokens is ultimately the external ledger that issued them. However, the source of truth for native tokens is the Ocean's ERC-1155 ledger. A native token's Ocean ID is determined upon creation when an Ocean-native primitive calls the public function `registerNewToken()`. The Ocean ID is a hash of the primitive's contract address and a nonce.

```
uint256 oceanID = uint256(keccak256(abi.encodePacked(primitiveAddress, nonce)));
```

3.3 Token Authority

In addition to the standard ERC-1155 mappings, the Ocean has one additional mapping, `tokensToPrimitives`, which maps a `uint256` (Ocean ID) to an address (primitive). This mapping determines which native token is under the "authority" of which external smart contract. Authority is given to the contract address of the primitive that creates the token.

Wrapped tokens do not have a formal entry in the `tokensToPrimitives` mapping because the external ledger has implicit authority over the wrapped token. Token authority is an important principle to understand when discussing the Ocean's accounting framework.

4 Accounting Framework for Ocean-Native Primitives

In order to understand the Ocean accounting framework, we must ask the question: What, exactly, is a financial primitive? We define a financial primitive as something that takes a token as input and then gives back another token as output. In other words, it converts one form of value into another form of value. In this way, a primitive is not so different from a market that exchanges one good for another.

4.1 Generalizing Accounting Logic to all Primitives

Consider what happens when Alice swaps 100 DAI for 99.9 USDC via an AMM (Table 1). If we look at the accounting, 100 DAI is subtracted from Alice's balance while being added to the AMM's balance, and 99.9 USDC is added to Alice's balance while being subtracted from the AMM's balance.

This pattern also holds when depositing into a lending pool (Table 2). Alice deposits 100 DAI into Aave, receiving aDAI in return (aDAI is an interest-bearing token that represents a claim on the underlying DAI). The one difference from the AMM swap is that aDAI is not subtracted from Aave's balance because Aave is the issuer of aDAI and instead mints new supply (i.e. Aave has "authority" over aDAI).

As seen in Table 3, we can generalize this transaction pattern. Token X is the "input token," which is added to the primitive's balance. Token Y is the "output token," which is added to Alice's balance. The "input amount" is Δx and the "output amount" is Δy. Alice can specify either the input amount or the output amount, but never both. The amount specified by Alice is referred to as the "specified amount." The amount computed by the primitive is referred to as the "computed amount."

	DAI Balance	***USDC Balance***
Alice	-100	$+99.9$
AMM	$+100$	-99.9

Table 1: AMM Accounting

	DAI Balance	***aDAI Balance***
Alice	-100	+100
Aave	+100	-0

Table 2: Lending Pool Accounting

	X Balance	***Y Balance***
Alice	$-\Delta x$	$+\Delta y$
Primitive	$+\Delta x$ or 0	$-\Delta y$ or 0

Table 3: Generalized Primitive Accounting

With this abstraction, we can deconstruct a primitive interaction into three steps:

1. Alice determines the input token (x), the output token (y) and specifies either the input amount (Δx) or the output amount (Δy).
2. The primitive determines the computed amount based on Alice's specified amount.
3. The input and output amounts (Δx, Δy) are added to and subtracted from the appropriate balances.

4.2 Encoding an Interaction

The Ocean's purpose is to execute primitive interactions as specified by Alice. Her instructions are encoded by the `Interaction` struct:

```
struct Interaction {
    bytes32 interactionTypeAndAddress;
    uint256 inputToken;
    uint256 outputToken;
    uint256 specifiedAmount;
    bytes32 metadata;
}
```

This struct encodes the information both for primitive interactions and for external ledger interactions. The `interactionTypeAndAddress` is a bytes array that contains both the interaction type and the address of the external contract with which Alice wants the Ocean to interact. The `interactionType` is a helper variable so that the Ocean can appropriately parse the struct. These two pieces of information, the transaction type and external contract address, are packed together in order to save gas. The `metadata` field is used for ERC-721 and ERC-1155 unwraps. It is also passed along to the primitive as a default, but it is up to the primitive how to use this field. For example, a lending pool may have Alice set her desired collateralization ratio in the `metadata` field. However, an AMM may not want to use `metadata` at all.

4.3 Executing an Interaction

Once encoded, the interaction is executed by the `_executeInteraction()` private function. This function is the primary workhorse of the Ocean. Figure 4 lays out the logic flow. For primitive interactions, `_executeInteraction()` queries the primitive for the computed amount, calling either `computeOutputAmount()` or `computeInputAmount()` depending on the `interactionType`. Once the primitive returns the computed amount, the Ocean mints and burns the input and output amounts from the associated balances as shown in Table 3.

The exception to this mint/burn pattern is if the primitive has issuing authority over the input and/or output tokens. In that case, it does not make sense to update the primitive's balance. Instead, the total supply of that token is increased for output tokens or decreased for input tokens. For example, when Alice deposits into an AMM, new LP tokens are minted and added to her balance. The Ocean does not deduct LP tokens from the AMM's balance.

4.4 Wrapping and Unwrapping Tokens

When interacting with external ledgers to wrap or unwrap tokens, the `_executeInteraction()` function follows the same pattern as with primitive interactions. The external ledger's contract address is encoded in the `interactionTypeAndAddress` bytes array. For ERC-721 and ERC-1155 tokens, the token ID is stored in the metadata field. For wraps, the specified amount is how many sh- tokens the user wants to mint. For unwraps, the specified amount is how many sh- tokens the user wants to burn.

To wrap a token, the Ocean calls the external ledger's `transferFrom()` function to receive the tokens from Alice. The Ocean then mints wrapped tokens to the user's balance. To unwrap a token, the Ocean burns wrapped tokens from Alice's balance and then calls the external ledger's `transfer()` function to give the tokens to Alice.

4.5 Using Mints and Burns instead of Transfers

Although subtle, the choice to move tokens via minting and burning instead of the standard ERC-1155 `transfer()` function is one of the largest differences between the Ocean's accounting and the status quo accounting practices in DeFi. There are two main advantages of this method:

- More flexibility
- Better gas efficiency

Any transfer can be decomposed into a mint and a burn; however, not every mint and burn can be reverse engineered as a transfer. In that way, minting and burning tokens is more fundamental than transferring tokens. For example, when depositing into an AMM, LP tokens will be minted, they will not be transferred. Moreover, minting and burning tokens also works for wrapping and unwrapping external tokens, whereas transfers do not. Hence, if the Ocean used transfers for primitive interactions, then there would be significantly more code complexity.

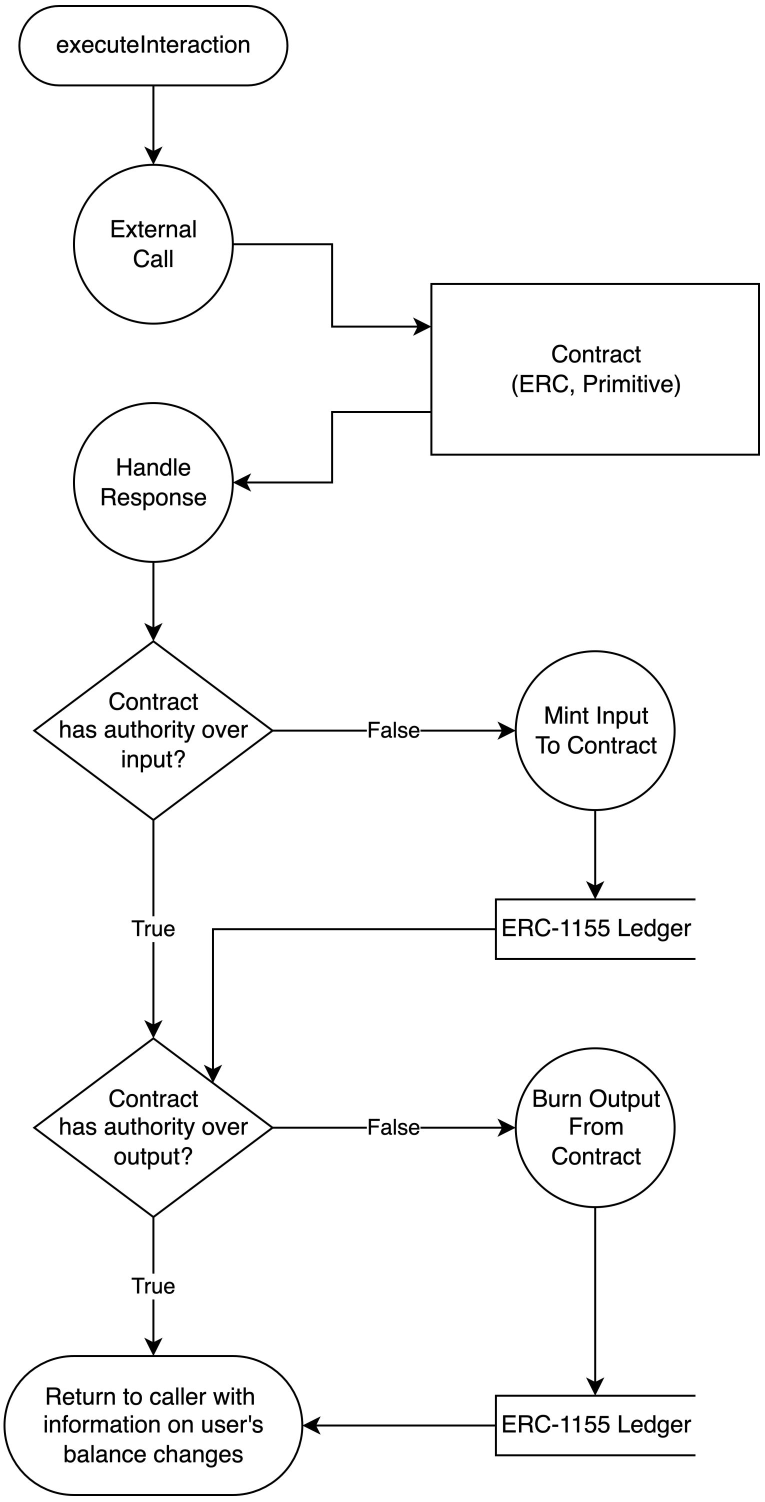

executeInteraction
External Call
Contract (ERC, Primitive)
Handle Response
Contract has authority over input?
False
Mint Input To Contract
True
ERC-1155 Ledger
Contract has authority over output?
False
Burn Output From Contract
True
Return to caller with information on user's balance changes
ERC-1155 Ledger

Figure 4: Abstract overview of `_executeInteraction`

The other major benefit of the mint/burn approach is that this method enables significant gas savings when chaining together a sequence of interactions, which will be the subject of the following section.

5 Composing Primitives on the Ocean

There are two basic public functions on the Ocean for Alice to interact with primitives:

- `doInteraction()`
- `doMultipleInteractions()`

The first function, `doInteraction()`, executes a single primitive interaction and contains minimal logic. The second function, `doMultipleInteractions()`, executes a sequence of primitive interactions and contains added logic to optimize the gas costs and provide more flexible accounting. The gas cost optimizations revolve around the difference between writing data to storage versus writing data to memory.

5.1 Tracking State Changes in Memory

As stated in Section 1.2, writing data to storage is one of the most expensive operations in the EVM, costing between 5,000 gas and 20,000 gas. Chaining primitive interactions together, such as swaps, can lead to unnecessary storage writes. For example, consider Alice swapping DAI to USDC, then swapping USDC to USDT. During this sequence, Alice's USDC balance first increases, then decreases with a net change of zero. With ERC-20 ledgers, we would end up with two storage writes that never needed to happen, not to mention the 2,600 gas cost from loading the ERC-20 contract bytecode.

In contrast, writing to memory is an order of magnitude cheaper; the base cost to write a `uint256` to memory is only 3 gas[10]. The single biggest difference between status quo accounting practices and the Ocean is that changes to Alice's balance during a sequence of interactions are tracked in memory, not storage. Balances are only updated in storage after the sequence concludes, avoiding redundant writes. In addition to saving gas, this approach gives Alice unprecedented flexibility.

5.2 `BalanceDelta`

To track Alice's balances in memory, the Ocean uses a struct called `BalanceDelta`:

```
struct BalanceDelta {
    uint256 tokenId;
    int256 delta;
}
```

The `tokenId` field is the token's Ocean ID. The `delta` field refers to the net change to Alice's balance. Note that because delta is an `int` rather than a `uint`, the net change can be negative. Although not immediately obvious, this feature allows the Ocean to natively support arbitrarily large flash mints[15] for tokens in its ERC-1155 ledger (see Appendix A).

At the start of the sequence, Alice passes to the Ocean an `Interaction[]` array. The Ocean then initializes a `BalanceDelta[]` array, with an entry for each token with which Alice will interact. All `delta` values are set to zero at the beginning. After a primitive returns the computed amount, the Ocean does not mint or burn tokens directly to Alice's balance in storage. Instead, the Ocean increments and decrements the `BalanceDelta[]` array, keeping a running tally of the net change to Alice's balances. The Ocean does not track changes to a primitive's balance in memory because doing so would not be worth the added complexity.

5.3 Writing State Changes to Storage

Once all interactions have been executed, the Ocean batch mints and burns tokens to Alice's account. Positive `delta` values are minted, negative are burned, and there are no storage writes for zero values. If the Ocean used transfers instead of mints and burns, then the `BalanceDelta` method would be infeasible. Figure 5 shows the logic flow of a multi-interaction sequence.

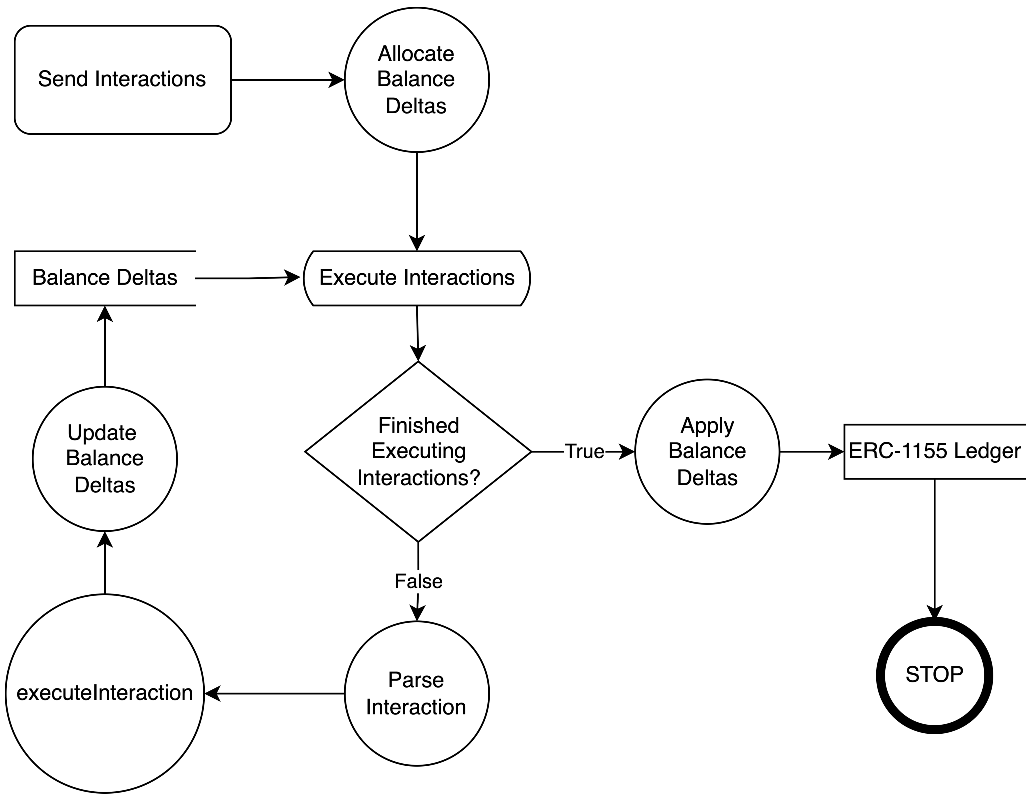
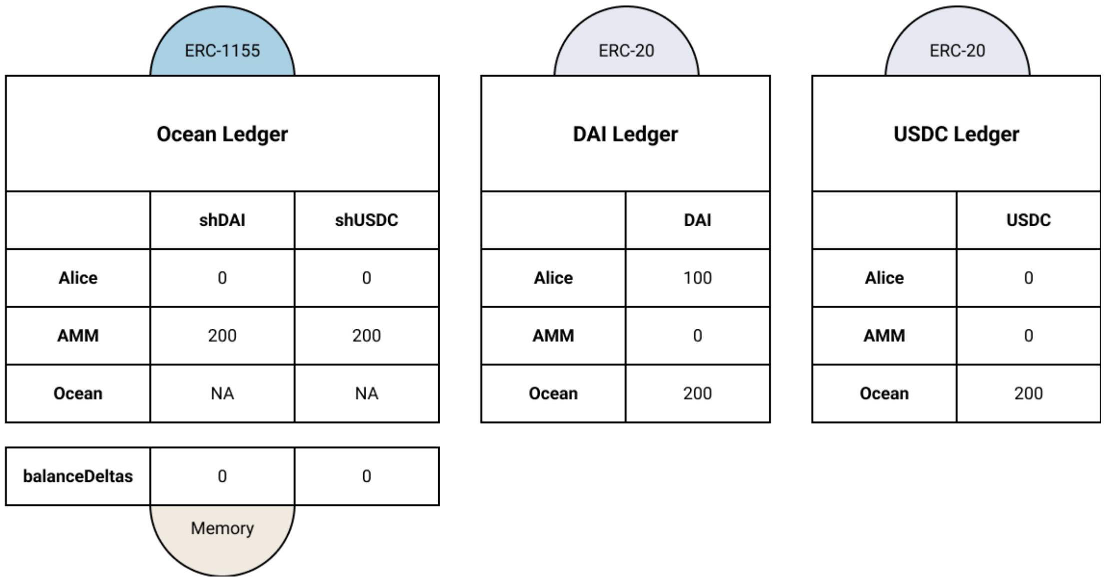


Figure 5: Abstract overview of `doMultipleInteractions()`

6 Example of a Composing Primitives on the Ocean

In this section we will demonstrate a simple end-to-end swap from DAI to USDC via an Ocean-native AMM. More complicated examples are provided in Appendix B as well. First, we will show the `Interaction[]` array. Then we will walk through the accounting.

At the start of the sequence, Alice has 100 DAI, but no USDC nor tokens held in the Ocean. Figure 6 shows the starting balances for Alice, the AMM and the Ocean. Note how the `BalanceDelta[]` values are initialized to zero.



Ocean Ledger

	shDAI	shUSDC
Alice	0	0
AMM	200	200
Ocean	NA	NA

balanceDeltas	0	0

DAI Ledger

	DAI
Alice	100
AMM	0
Ocean	200

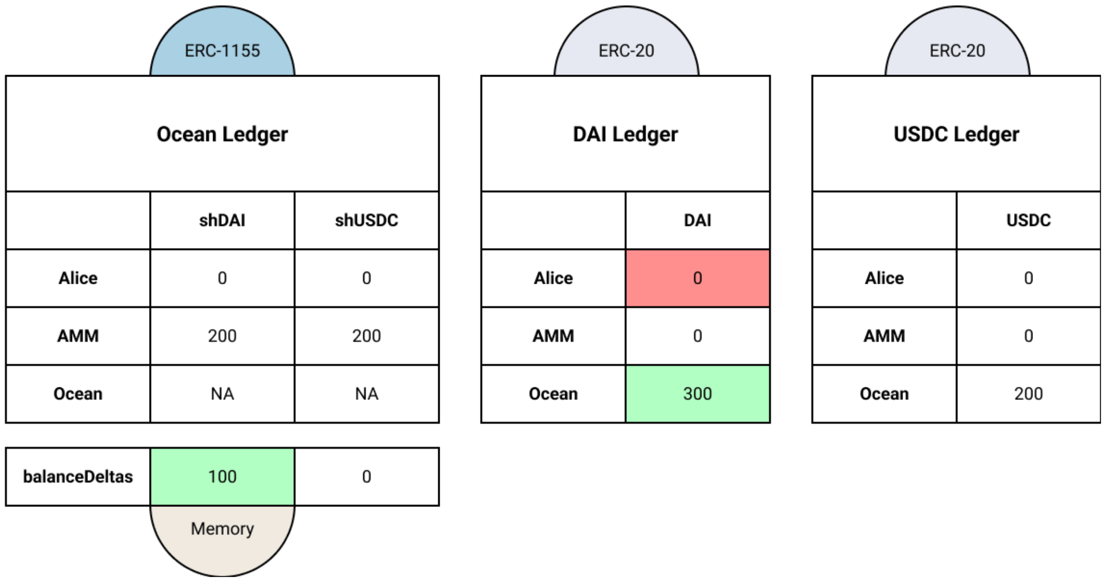
USDC Ledger

	USDC
Alice	0
AMM	0
Ocean	200

Figure 6: Initial state of the ledgers before the first interaction

The first interaction wraps 100 DAI into the Ocean. We specified the amount in 18 decimals because that is the convention for all fungible Ocean tokens. Alice is converting her ERC-20 DAI to the Ocean's ERC-1155 shDAI. Figure 7 shows the intermediate balances after the interaction. Alice's shDAI is tracked in the `BalanceDelta[]` array.

```
Interaction wrap {
    bytes32 interactionTypeAndAddress = {WrapErc20, DAI};
    uint256 inputToken = 0;
    uint256 outputToken = 0;
    uint256 specifiedAmount = 100*10^18;
    bytes32 metadata = 0;
}
```



ERC-1155

Ocean Ledger

	shDAI	shUSDC
Alice	0	0
AMM	200	200
Ocean	NA	NA

balanceDeltas	🟩 100	0

Memory

ERC-20

DAI Ledger

	DAI
Alice	🟥 0
AMM	0
Ocean	🟩 300

ERC-20

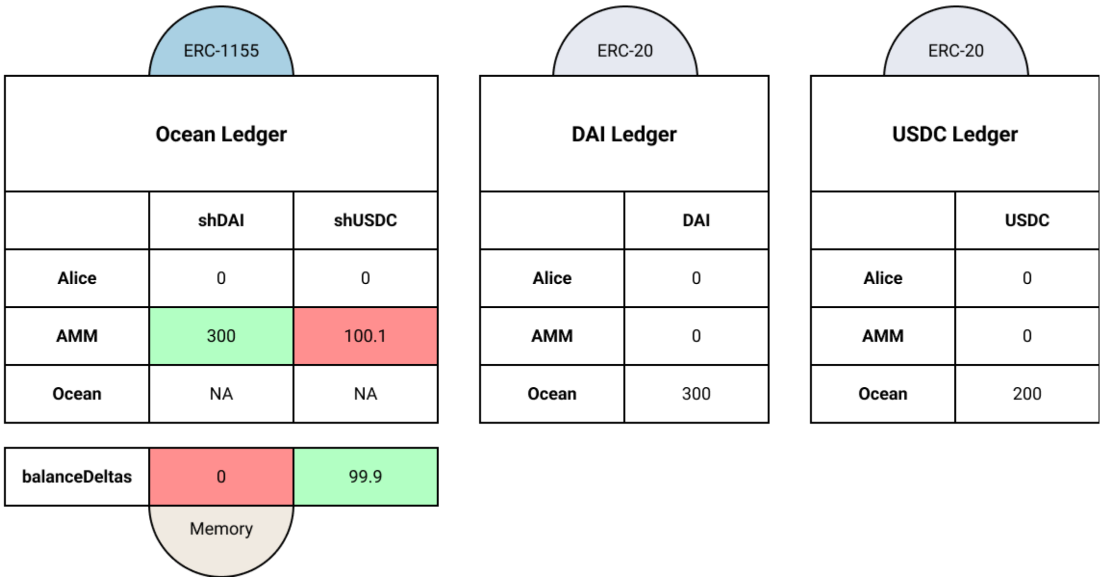
USDC Ledger

	USDC
Alice	0
AMM	0
Ocean	200

Figure 7: Interaction 1: Alice wraps 100 DAI into the Ocean. (Red indicates a decrease; green indicates an increase)

The second interaction swaps 100 shDAI for 99.9 shUSDC (Figure 8). Interaction type 7 denotes an Ocean-native primitive interaction where Alice specifies the input amount.

```
Interaction swap {
    Bytes32 interactionTypeAndAddress = {ComputeOutputAmount, AMM};
    uint256 inputToken = shDAI;
    uint256 outputToken = shUSDC;
    uint256 specifiedAmount = 100*10^18;
    bytes32 metadata = 0;
}
```



Ocean Ledger	shDAI	shUSDC
Alice	0	0
AMM	300	100.1
Ocean	NA	NA

DAI Ledger	DAI
Alice	0
AMM	0
Ocean	300

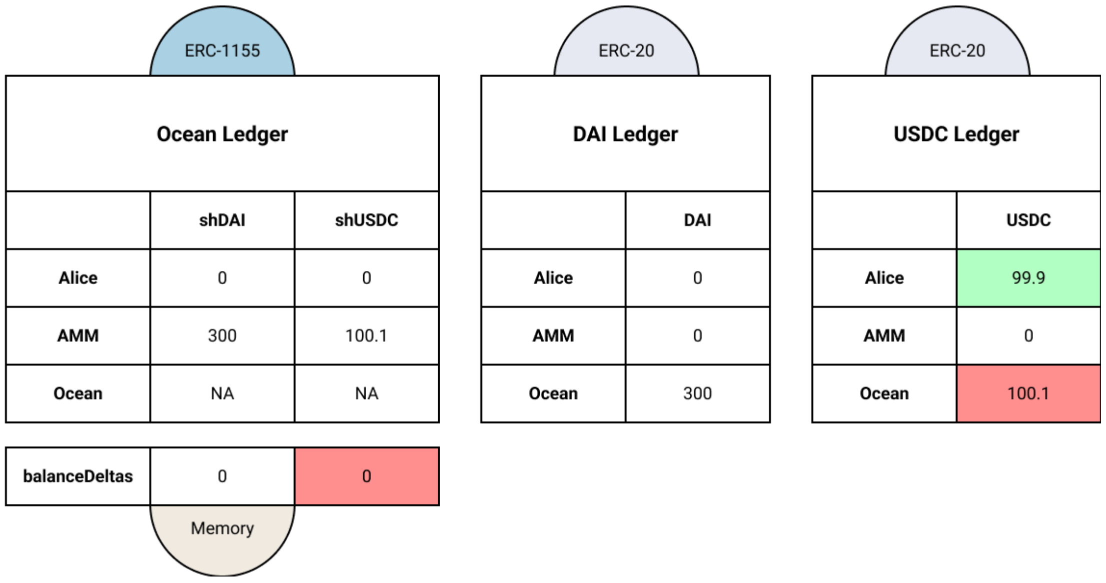
USDC Ledger	USDC
Alice	0
AMM	0
Ocean	200

balanceDeltas	0	99.9

Figure 8: Interaction 2: Alice swaps 100 shDAI for 99.9 shUSDC

The third interaction unwraps USDC from the Ocean and sends it to Alice. Interaction type 1 denotes an ERC-20 unwrap. The `specifiedAmount`, set to `max`, is interpreted by the Ocean as an opcode. The Ocean queries the `BalanceDelta[]` array and substitutes the current `delta` value as the specified amount. We use the `max` opcode because when we are generating the `Interaction[]` array, we do not know the exact swap rate between shDAI and shUSDC. Thus we do not know the exact amount of shUSDC we will need to unwrap. However, we can count on the `BalanceDelta` tracking the exact amount remaining. By instructing the Ocean to use the remainder, Alice is guaranteed not to be holding small amounts of shUSDC at the end of the transaction. Figure 9 shows the state of the ledger.

```
Interaction unwrap {
    bytes32 interactionTypeAndAddress = {UnwrapErc20, USDC};
    uint256 inputToken = 0;
    uint256 outputToken = 0;
    uint256 specifiedAmount = max;
    bytes32 metadata = 0;
}
```


ERC-1155

Ocean Ledger

	shDAI	shUSDC
Alice	0	0
AMM	300	100.1
Ocean	NA	NA

balanceDeltas	0	0

Memory

ERC-20

DAI Ledger

	DAI
Alice	0
AMM	0
Ocean	300

ERC-20

USDC Ledger

	USDC
Alice	99.9
AMM	0
Ocean	100.1

Figure 9: Interaction 3: Alice unwraps 99.9 USDC out of the Ocean

Lastly, the Ocean reconciles the `BalanceDelta[]` array with Alice's balance in storage. In this case, there are no net changes to Alice's Ocean ERC-1155 balances, hence there are no storage writes.

7 Ocean Performance

As stated in Section 2.1, the first criterion with which to assess the Ocean is gas savings. The second is how much additional smart contract code, if any, is needed to compose primitives. For the latter, the Ocean has built-in logic to compose Ocean-native primitives (see Section 5), so it's clear that no additional smart contract code is necessary. Hence, this section will focus entirely on gas savings.

There are some initial caveats to address. A lot of variables go into determining gas costs[10]. For example, writing a new `uint256` to storage costs 20,000 gas, while overwriting an existing `uint256` costs only 5,000 gas (see Section 1.2). In all of our tests, we will calculate gas based on over-writing existing storage whenever possible.

We can break down gas costs into two parts: the fixed cost and the marginal cost. The fixed cost is the cost to execute a single primitive interaction. The marginal cost is the amount the cost increases for each subsequent primitive interaction. For example, if interacting with one primitive costs 100,000 gas while interacting with two primitives costs 150,000 gas, then the fixed cost would be 100,000 gas and the marginal cost for the second interaction would be 50,000 gas. When composing one or two primitives, the fixed cost will be more important to the total cost. However, as we compose more primitives, the marginal cost will start to dominate.

We will conduct three tests, swapping two tokens through the Ocean via Ocean-native AMMs.

We will use a constant sum AMM[5], keeping the bonding curve simple in order to focus our results on the Ocean per se, and not the AMM logic. Our first test will use the Ocean, without any of its interaction sequencing logic, to serve as a baseline for comparison. This will emulate the status quo method of composing primitives in DeFi. Each swap will involve its own function call to the Ocean, with a wrap and unwrap, going through a different Ocean-native AMM each time.

Our second and third tests will take full advantage of the Ocean's interaction sequencing logic. The second test will split the swap through n different Ocean-native AMMs, with only one wrap and unwrap. The third test will chain swaps through n different AMMs. The output from AMM i will be the input for AMM $i+1$.

Figure 10 shows the marginal costs of each test and Figure 11 shows the total cost. We can see that the marginal cost of both Test 2 and Test 3 are substantially lower than the baseline. Averaging between the two, the marginal cost is approximately four times lower when we compose primitives on the Ocean versus using the status quo method. As a result, the total cost of composing 10 primitives is also substantially lower. Averaging Test 2 and Test 3, the total cost after the tenth interaction is three times lower than the baseline. Any primitive deployed onto the Ocean will automatically inherit these gas savings.

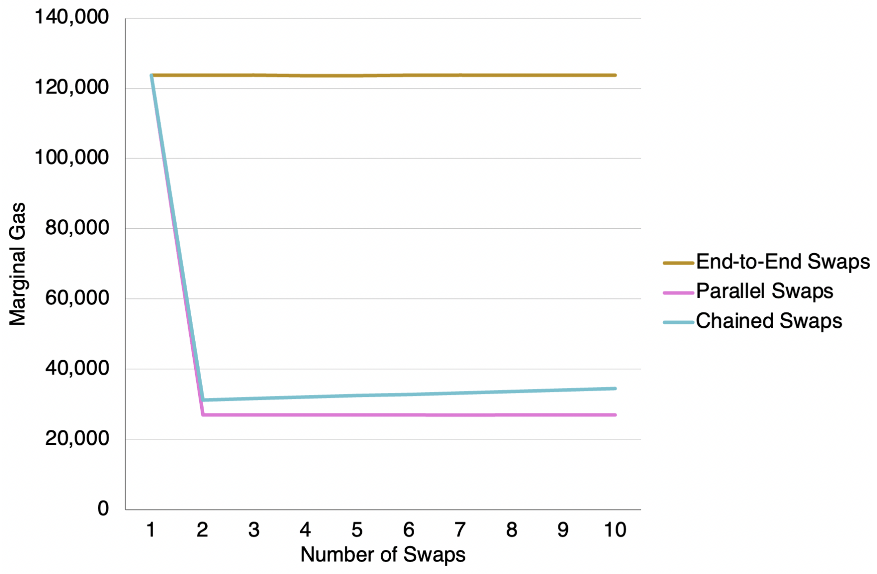


Figure 10: Marginal cost per primitive interaction

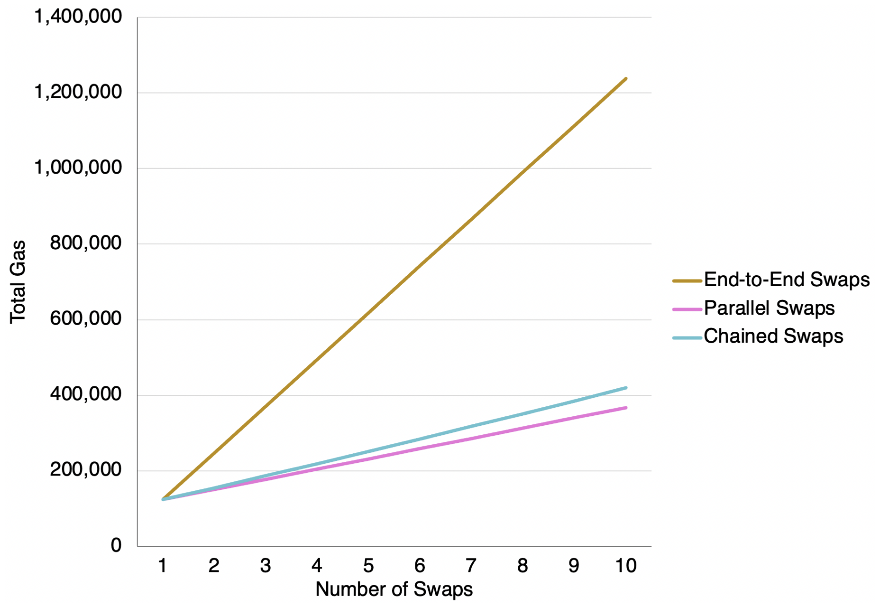


Figure 11: Total cost per primitive interaction

8 Ocean Benefits

The Ocean and the ecosystem that grows around it will bring significant benefits to end users, developers of Web3 applications, and developers of new primitives. For end users, the Ocean will improve UX by saving substantial amounts of gas and eliminating the time lost while waiting for multiple transactions to process. In short, they will save both time and money.

Instead of separately integrating with every primitive, Web3 application developers will only need to integrate with one smart contract: the Ocean. They will then have access to every Ocean-native primitive. As more primitives join the ecosystem, Web3 developers can seamlessly incorporate them into their application.

More importantly, these developers can now compose primitives without needing to write or deploy specialized smart contracts. No Solidity required. By reducing the barriers for Web3 apps to compose primitives, developers can offer increasingly flexible and complex financial services to their users. For example, NFT markets built on the Ocean can leverage liquidity from Ocean-native AMMs so that users can become currency agnostic, meaning they can buy and sell NFTs using any currency they choose, ETH, USDC, DAI, etc.

Perhaps the biggest beneficiaries will be teams building new primitives. Ocean-native primitives are simpler to design because they only need to implement the business logic; the Ocean will handle the accounting logic. Additionally, by connecting to the Ocean, primitives will become part of a larger network. When two or more primitives are composed together, both primitives benefit. For example, when an AMM is composed with a lending pool to offer leverage, the AMM gets more trade volume and the lending pool gets more deposits.

The more primitives that join the network, the more options for composability, and the more each individual primitive benefits. There is a multiplicative network effect. To put things another way: would you rather be a lonesome tree in the middle of the desert? Or would you rather be a tree in the rainforest, surrounded by a vibrant ecosystem?

When the Ocean first launches, the ecosystem will start out empty. However, Ocean-native primitives will be at no disadvantage versus non-native primitives when it comes to gas costs and ease of composability. Furthermore, Shell Protocol will employ tokenomic mechanisms to bootstrap growth of the network.

9 Conclusion

Composing primitives is difficult because the system architecture of DeFi is muddled by siloed token ledgers and monolithic financial primitives. The Ocean is a new paradigm for DeFi that is designed to seamlessly and efficiently compose any type of primitive: AMMs, lending pools, algorithmic stablecoins, NFT markets, or even primitives yet to be invented. Composing primitives on the Ocean can save up to four times the marginal gas cost and requires no additional smart contracts beyond the core protocol. Not only are primitives built on the Ocean simpler, they also become part of a larger, composable ecosystem.

References

[1] 1inch Network (2022). “1inch Aggregation Protocol.” https://1inch.io/aggregation-protocol/

[2] Aave (2022, January 21). “Aave FAQs.” https://docs.aave.com/faq/

[3] Adams, Hayden (2019). “Uniswap White Paper.” https://hackmd.io/@HaydenAdams/HJ9jLsfTz

[4] Adams et al (March, 2021). “Uniswap v3 Core.” https://uniswap.org/whitepaper-v3.pdf

[5] Berenzon, Dmitriy (2020, April). “Constant Function Market Makers: DeFi’s “Zero to One” Innovation.” https://medium.com/bollinger-investment-group/constant-function-market-makers-defis-zero-to-one-innovation-968f77022159

[6] Buterin, Vitalik and Martin Swende (2020). “EIP-2929: Gas cost increases for state access opcodes.” https://eips.ethereum.org/EIPS/eip-2929

[7] Egorov, Michael (2019, November 10). “StableSwap - efficient mechanism for Stablecoin liquidity.” https://curve.fi/files/stableswap-paper.pdf

[8] Entriken et al (2018). “EIP-721: Non-Fungible Token Standard.” https://eips.ethereum.org/EIPS/eip-721

[9] Ethereum Foundation (2022). “Ethereum Virtual Machine (EVM).” https://ethereum.org/en/developers/docs/evm/

[10] Ethereum Foundation (2022). “Opcodes for the EVM.” https://ethereum.org/en/developers/docs/evm/opcodes/

[11] Hertig, Alyssa (2021, February 17). “What Is a Flash Loan? A guide to one of DeFi’s most innovative and controversial features.” https://www.coindesk.com/learn/2021/02/17/what-is-a-flash-loan/

[12] InstaDapp (2022). “InstaDapp Docs: Overview.” https://docs.instadapp.io/

[13] Kunz, Sergej (2021). “What Are DEX Aggregators? A Deep Dive by 1inch.” https://coinmarketcap.com/alexandria/article/what-are-dex-aggregators-a-deep-dive-by-1inch

[14] Leshner, Robert and Geoffrey Hayes (2019, February). “Compound: The Money Market Protocol”. https://compound.finance/documents/Compound.Whitepaper.pdf

[15] MacPherson, Sam (2020, September). “MIP25: Flash Mint Module.” https://forum.makerdao.com/t/mip25-flash-mint-module/4400

[16] Maker DAO (2020). “The Maker Protocol: MakerDAO’s Multi-Collateral Dai (MCD) System.” https://makerdao.com/en/whitepaper/#abstract

[17] Martinelli, Fernando (2021, Feb 2). “Introducing Balancer V2: Generalized AMMs.” https://medium.com/balancer-protocol/balancer-v2-generalizing-amms-16343c4563ff

[18] OpenSea (2021). “OpenSea Developer Platform.” https://docs.opensea.io/

[19] Paraswap (2022). “Paraswap FAQs.” https://doc.paraswap.network/getting-started/faq

[20] Radomski et al (2018). “EIP-1155: Multi Token Standard.” https://eips.ethereum.org/EIPS/eip-1155

[21] Shell Protocol (2021, October). “Proteus AMM Engine.” https://github.com/cowri/Proteus/blob/main/Proteus_AMM_Engine_-_Shell_v2_Part_1.pdf

[22] Tian, Ryan (2021). “A Deep Dive Into Leverages in DeFi Borrowing, Margin Trading, Leveraged Tokens and Options: FinNexus.” https://coinmarketcap.com/alexandria/article/a-deep-dive-into-leverages-in-defi-borrowing-margin-trading-leveraged-tokens-and-options-finnexus

[23] Totle (2019, August 16). “Building with Money Legos.” https://medium.com/totle/building-with-money-legos-ab63a58ae764

[24] Vogelsteller, Fabian and Vitalik Buterin (2015). “EIP-20: Token Standard.” https://eips.ethereum.org/EIPS/eip-20

[25] Wikipedia (2022). “Mansa Musa.” https://en.wikipedia.org/wiki/Mansa_Musa

[26] Zapper (2022). “Zapper Docs: Introduction.” https://docs.zapper.fi/

A Flash Loans on the Ocean

As noted previously, the BalanceDelta[] array can have negative entries. In that way, this temporary ledger serves as a debit and credit system during the interaction sequence. As a consequence, Alice can spend money she doesn't have, provided she can make up the difference by the end of the sequence (see Appendix B.3 for an example).

Although not immediately obvious, the Ocean natively supports arbitrarily large flash mints[15] for tokens in its ERC-1155 ledger. The limit is only constrained by the size of an int256, which is roughly equal to the number of particles in the observable universe. Anyone in the world can be an ephemeral Mansa Musa[25].

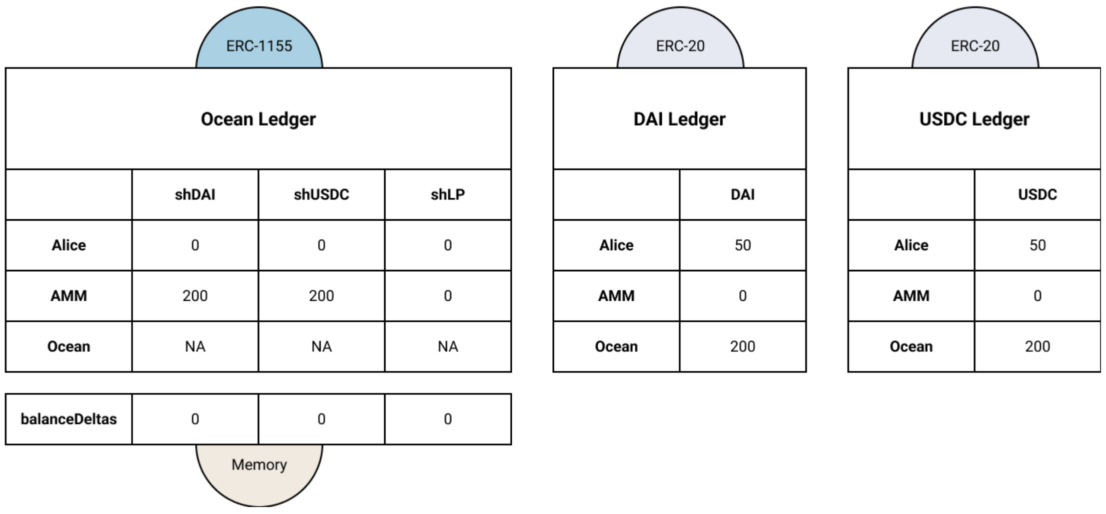
Because flash mints are perfunctory, arbitraging prices between Ocean-native primitives is extremely gas efficient and relatively easy. No special smart contracts are needed. In-Ocean flash mints can also be unwrapped, resulting in an external flash loan[11], a cheap source of capital to be used outside the protocol. To be clear, external flash loans are constrained by the amount of tokens wrapped in the Ocean. Flash mints only apply to the internal ledger.

B Examples of `doMultipleInteractions()`

The Ocean is extremely flexible and thus far we have focused on how it works and the gas efficiency, but not what it is capable of. This section will provide more examples of what is possible to do on the Ocean.

B.1 LP into an AMM

In this example, Alice will deposit DAI and USDC into an AMM on the Ocean, minting LP tokens. Before processing the interactions, the Ocean initializes the `BalanceDelta[]` array (Figure 12).



Ocean Ledger	shDAI	shUSDC	shLP
Alice	0	0	0
AMM	200	200	0
Ocean	NA	NA	NA
balanceDeltas	0	0	0

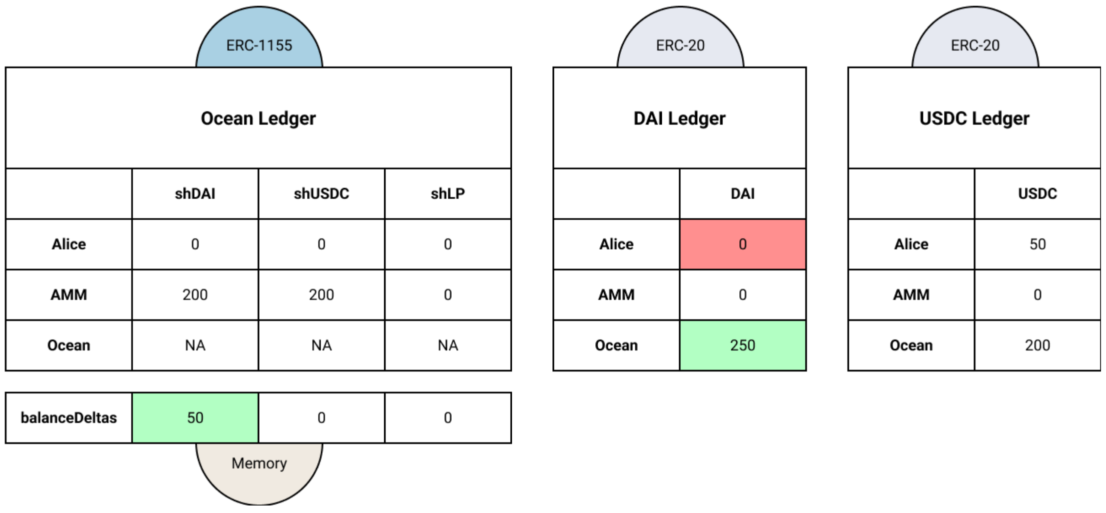
DAI Ledger	DAI
Alice	50
AMM	0
Ocean	200

USDC Ledger	USDC
Alice	50
AMM	0
Ocean	200

Figure 12: Initial state of the ledger before LP

In the first interaction, Alice wraps DAI into the Ocean, minting shDAI (Figure 13).

```
Interaction wrapDAI {
    bytes32 interactionTypeAndAddress = {WrapErc20, DAI};
    uint256 inputToken = 0;
    uint256 outputToken = 0;
    uint256 specifiedAmount = 50*10^18;
    bytes32 metadata = 0;
}
```



Ocean Ledger	shDAI	shUSDC	shLP
Alice	0	0	0
AMM	200	200	0
Ocean	NA	NA	NA
balanceDeltas	50	0	0

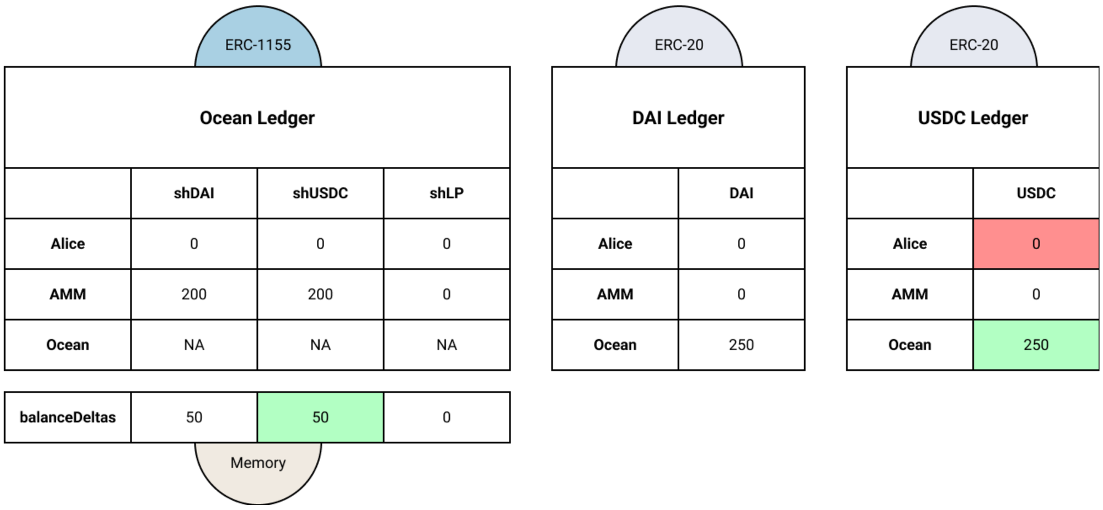
DAI Ledger	DAI
Alice	0
AMM	0
Ocean	250

USDC Ledger	USDC
Alice	50
AMM	0
Ocean	200

Figure 13: Interaction 1: Alice wraps 50 DAI into the Ocean

In the second interaction, Alice wraps USDC, minting shUSDC (Figure 14).

```
Interaction wrapUSDC {
    bytes32 interactionTypeAndAddress = {WrapErc20, USDC};
    uint256 inputToken = 0;
    uint256 outputToken = 0;
    uint256 specifiedAmount = 50*10^18;
    bytes32 metadata = 0;
}
```



Ocean Ledger

	shDAI	shUSDC	shLP
Alice	0	0	0
AMM	200	200	0
Ocean	NA	NA	NA

balanceDeltas	50	🟩 50	0

DAI Ledger

	DAI
Alice	0
AMM	0
Ocean	250

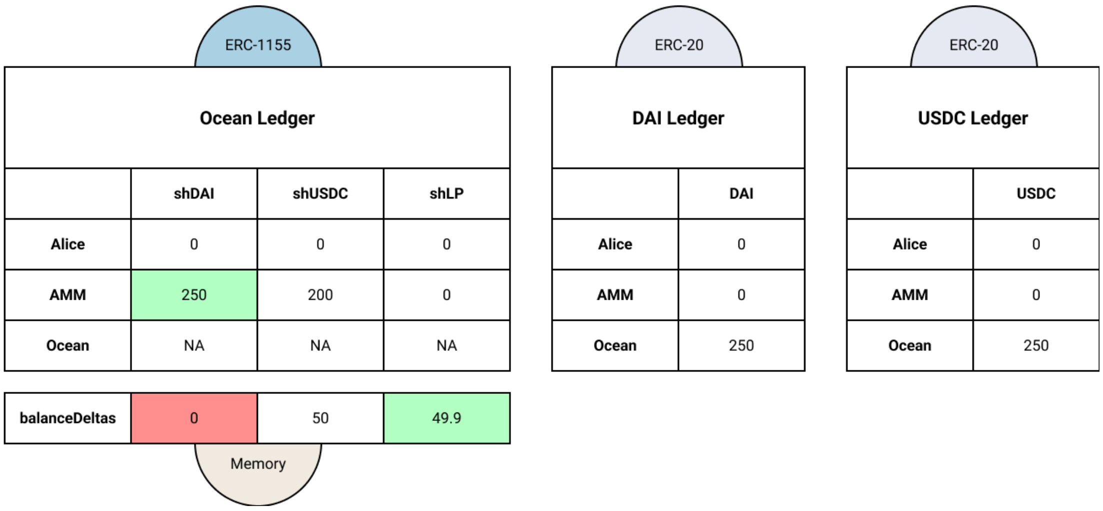
USDC Ledger

	USDC
Alice	🟥 0
AMM	0
Ocean	🟩 250

Figure 14: Interaction 2: Alice wraps 50 USDC into the Ocean

In the third interaction, Alice deposits shDAI into the AMM, minting LP tokens (Figure 15).

```
Interaction depositDAI {
    bytes32 interactionTypeAndAddress = {ComputeOutputAmount, AMM};
    uint256 inputToken = shDAI;
    uint256 outputToken = LP;
    uint256 specifiedAmount = 50*10^18;
    bytes32 metadata = 0;
}
```



Ocean Ledger

	shDAI	shUSDC	shLP
Alice	0	0	0
AMM	250	200	0
Ocean	NA	NA	NA
balanceDeltas	0	50	49.9

DAI Ledger

	DAI
Alice	0
AMM	0
Ocean	250

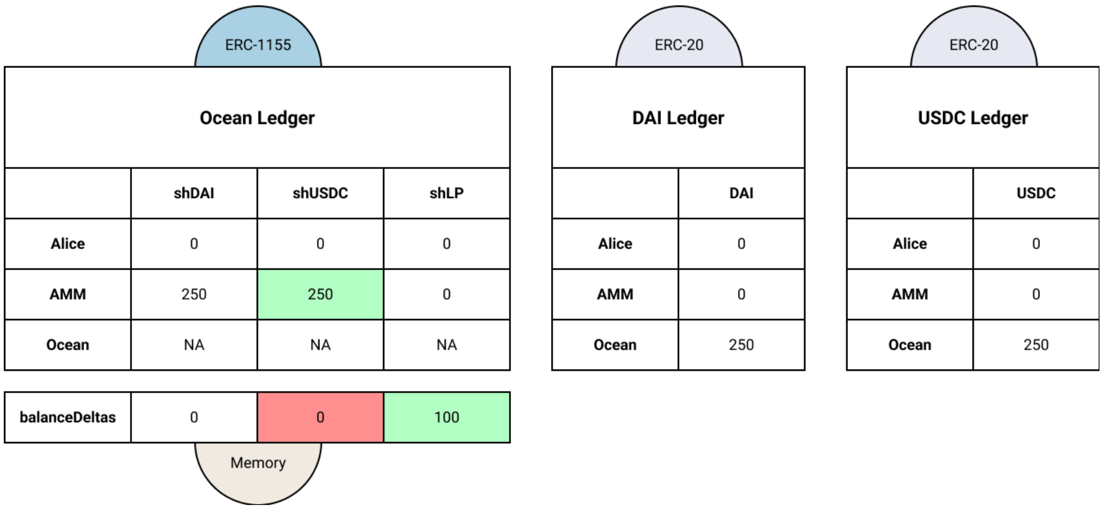
USDC Ledger

	USDC
Alice	0
AMM	0
Ocean	250

Figure 15: Interaction 3: Alice deposits 50 shDAI into the Ocean

In the fourth and final interaction, Alice deposits shUSDC into the AMM, minting LP tokens (Figure 16).

```
Interaction depositUSDC {
    bytes32 interactionTypeAndAddress = {ComputeOutputAmount, AMM};
    uint256 inputToken = shUSDC;
    uint256 outputToken = LP;
    uint256 specifiedAmount = 50*10^18;
    bytes32 metadata = 0;
}
```



Ocean Ledger

	shDAI	shUSDC	shLP
Alice	0	0	0
AMM	250	250	0
Ocean	NA	NA	NA
balanceDeltas	0	0	100

DAI Ledger

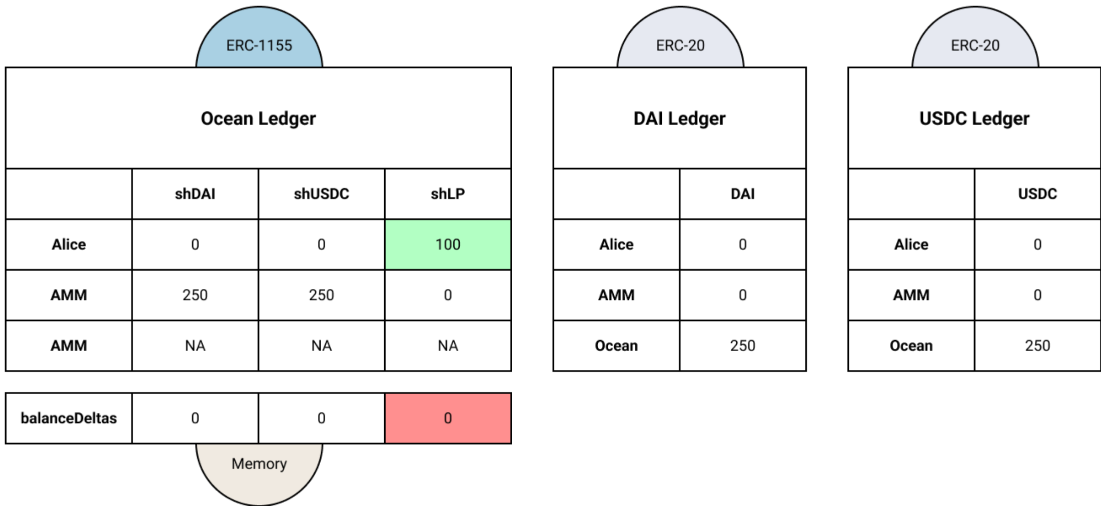
	DAI
Alice	0
AMM	0
Ocean	250

USDC Ledger

	USDC
Alice	0
AMM	0
Ocean	250

Figure 16: Interaction 4: Alice deposits 50 shUSDC into the Ocean

Once all interactions have been processed, the Ocean writes the `BalanceDelta[]` array to storage in the Ocean's ERC-1155 ledger (Figure 17).



ERC-1155 — Ocean Ledger

	shDAI	shUSDC	shLP
Alice	0	0	100
AMM	250	250	0
AMM	NA	NA	NA

balanceDeltas	0	0	0

ERC-20 — DAI Ledger

	DAI
Alice	0
AMM	0
Ocean	250

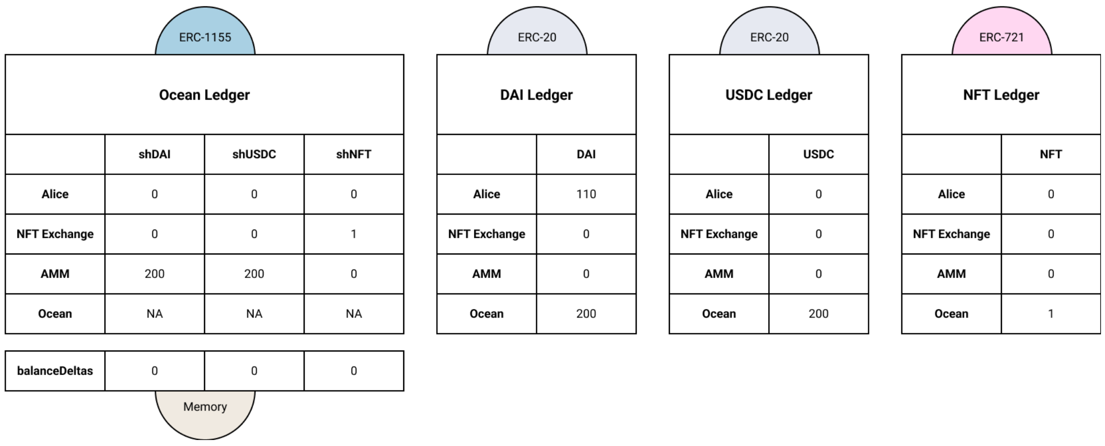
ERC-20 — USDC Ledger

	USDC
Alice	0
AMM	0
Ocean	250

Figure 17: Final state of the ledger after all interactions have been executed.

B.2 Purchasing an NFT after a Swap

In this example, Alice will transact in the opposite direction from the previous examples. Instead of specifying how much she will give to the primitives, Alice will specify how much she wants to receive. Because the `BalanceDelta` can hold negative values, this type of transaction pattern is feasible. Before processing the interactions, the Ocean initializes the `BalanceDelta[]` array (Figure 18).



ERC-1155 — Ocean Ledger

	shDAI	shUSDC	shNFT
Alice	0	0	0
NFT Exchange	0	0	1
AMM	200	200	0
Ocean	NA	NA	NA

balanceDeltas	0	0	0

ERC-20 — DAI Ledger

	DAI
Alice	110
NFT Exchange	0
AMM	0
Ocean	200

ERC-20 — USDC Ledger

	USDC
Alice	0
NFT Exchange	0
AMM	0
Ocean	200

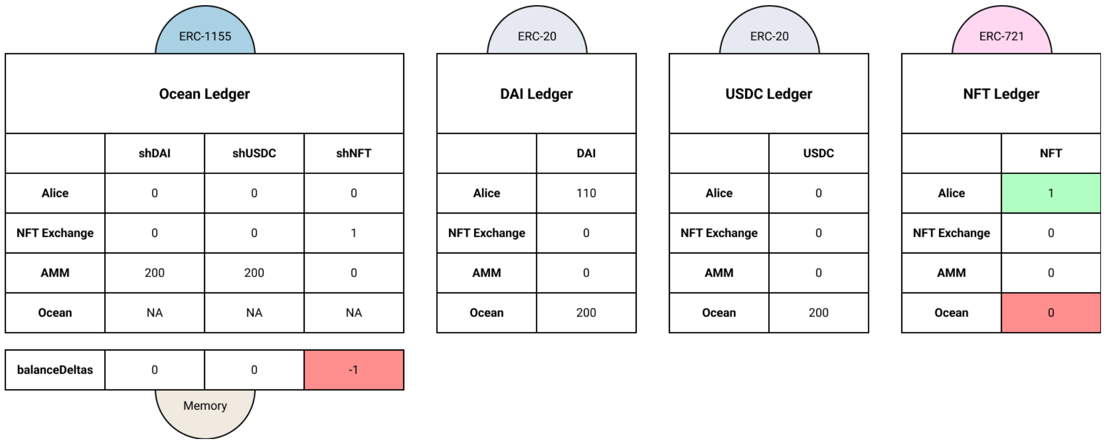
ERC-721 — NFT Ledger

	NFT
Alice	0
NFT Exchange	0
AMM	0
Ocean	1

Figure 18: Initial state of the ledger before NFT purchase

In the first interaction, Alice unwraps the NFT from the Ocean (Figure 19). Alice is able to do this because the BalanceDelta can go negative.

```
Interaction unwrapNFT {
    bytes32 interactionTypeAndAddress = {UnwrapErc721, NFT};
    uint256 inputToken = 0;
    uint256 outputToken = 0;
    uint256 specifiedAmount = 1;
    bytes32 metadata = tokenID;
}
```



ERC-1155

Ocean Ledger

	shDAI	shUSDC	shNFT
Alice	0	0	0
NFT Exchange	0	0	1
AMM	200	200	0
Ocean	NA	NA	NA
balanceDeltas	0	0	-1

Memory

ERC-20

DAI Ledger

	DAI
Alice	110
NFT Exchange	0
AMM	0
Ocean	200

ERC-20

USDC Ledger

	USDC
Alice	0
NFT Exchange	0
AMM	0
Ocean	200

ERC-721

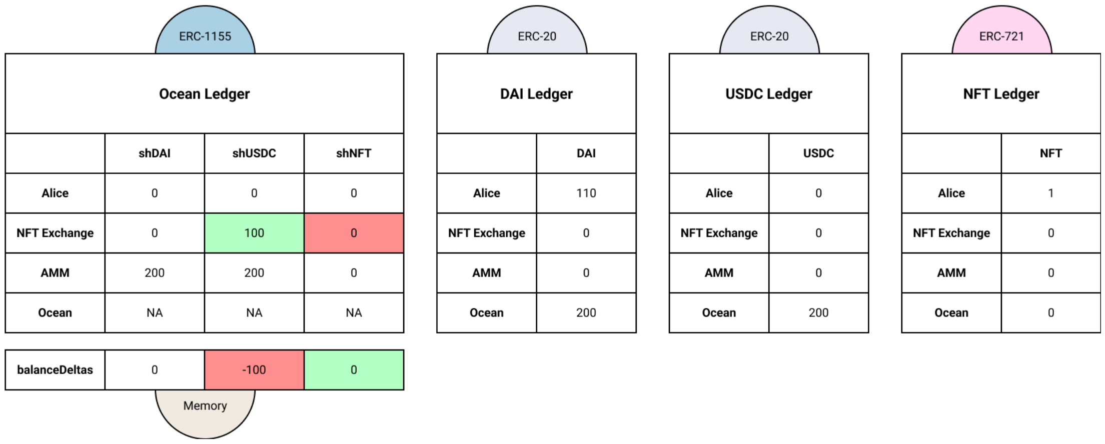
NFT Ledger

	NFT
Alice	1
NFT Exchange	0
AMM	0
Ocean	0

Figure 19: Interaction 1: Alice unwraps the NFT from the Ocean

In the second interaction, Alice purchases the NFT from the exchange using USDC (Figure 20). She does not specify how much USDC she will give to the exchange, but instead specifies that she wants to purchase the NFT. The exchange then sets the price.

```
Interaction purchaseNFT {
    bytes32 interactionTypeAndAddress = {ComputeInputAmount, NFTMarket};
    uint256 inputToken = DAI;
    uint256 outputToken = NFT;
    uint256 specifiedAmount = 1;
    bytes32 metadata = 0;
}
```


ERC-1155
ERC-20
ERC-20
ERC-721
Memory

Ocean Ledger	shDAI	shUSDC	shNFT
Alice	0	0	0
NFT Exchange	0	100	0
AMM	200	200	0
Ocean	NA	NA	NA
balanceDeltas	0	-100	0

DAI Ledger	DAI
Alice	110
NFT Exchange	0
AMM	0
Ocean	200

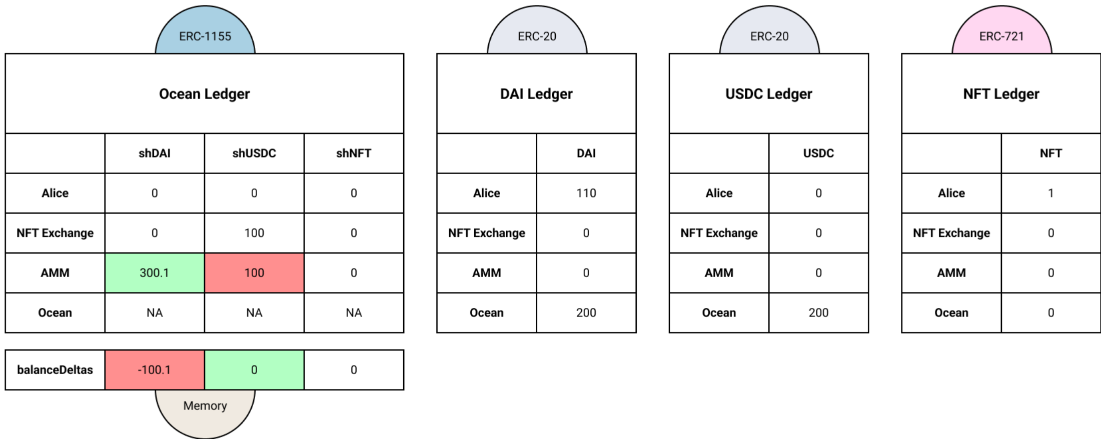
USDC Ledger	USDC
Alice	0
NFT Exchange	0
AMM	0
Ocean	200

NFT Ledger	NFT
Alice	1
NFT Exchange	0
AMM	0
Ocean	0

Figure 20: Interaction 2: Alice purchases the NFT from the exchange using USDC

In the third interaction, Alice swaps DAI for USDC (Figure 21). She specifies the amount of USDC she would like to purchase (rather than how much DAI she would like to swap). She uses the `max` opcode to instruct the Ocean to use her outstanding USDC debit held in the `BalanceDelta[]` array. She does this because it guarantees that she won't have any outstanding debits or credits to her shUSDC balance at the end of the transaction sequence.

```
Interaction swap {
    bytes32 interactionTypeAndAddress = {ComputeInputAmount, AMM};
    uint256 inputToken = USDC;
    uint256 outputToken = DAI;
    uint256 specifiedAmount = max;
    bytes32 metadata = 0;
}
```


ERC-1155
ERC-20
ERC-20
ERC-721
Memory

Ocean Ledger	shDAI	shUSDC	shNFT
Alice	0	0	0
NFT Exchange	0	100	0
AMM	300.1	100	0
Ocean	NA	NA	NA
balanceDeltas	-100.1	0	0

DAI Ledger	DAI
Alice	110
NFT Exchange	0
AMM	0
Ocean	200

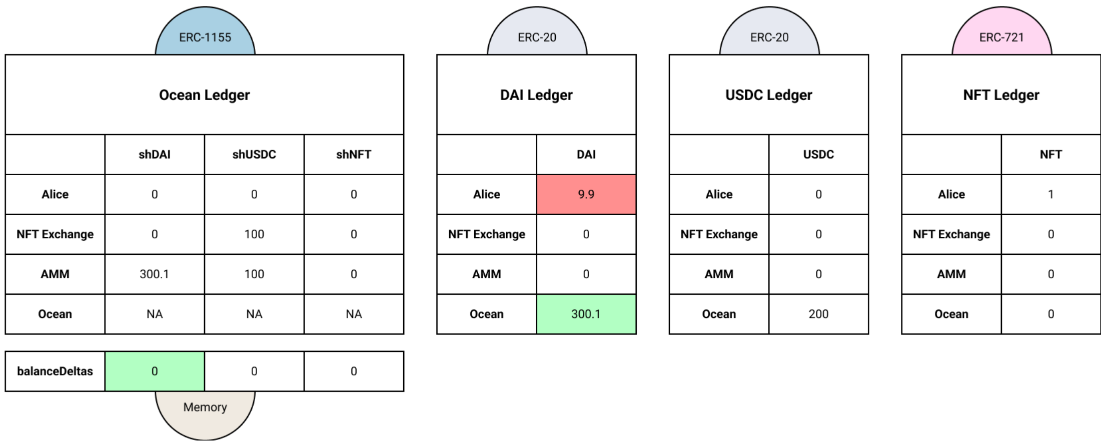
USDC Ledger	USDC
Alice	0
NFT Exchange	0
AMM	0
Ocean	200

NFT Ledger	NFT
Alice	1
NFT Exchange	0
AMM	0
Ocean	0

Figure 21: Interaction 3: Alice swaps DAI for USDC

In the fourth and final interaction, Alice wraps her DAI into the Ocean in order to pay for the swap (Figure 22). She uses the `max` opcode because she will not know for certain the exact amount of DAI she had to swap in the previous interaction. This ensures that she won't have any outstanding debits or credits to her shDAI balance at the end of the transaction sequence.

```
Interaction wrapDAI {
    bytes32 interactionTypeAndAddress = {WrapErc20, DAI};
    uint256 inputToken = 0;
    uint256 outputToken = 0;
    uint256 specifiedAmount = max;
    bytes32 metadata = 0;
}
```



Ocean Ledger

	shDAI	shUSDC	shNFT
Alice	0	0	0
NFT Exchange	0	100	0
AMM	300.1	100	0
Ocean	NA	NA	NA

balanceDeltas	0	0	0

DAI Ledger

	DAI
Alice	9.9
NFT Exchange	0
AMM	0
Ocean	300.1

USDC Ledger

	USDC
Alice	0
NFT Exchange	0
AMM	0
Ocean	200

NFT Ledger

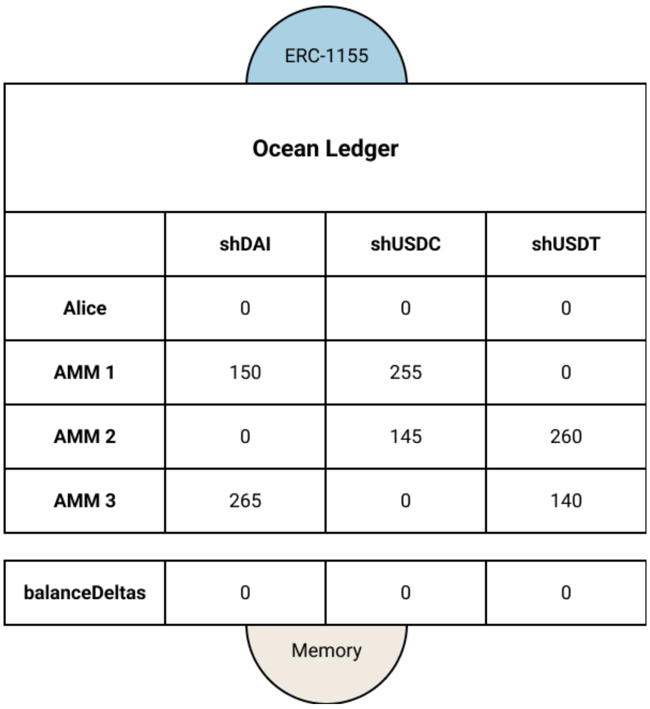
	NFT
Alice	1
NFT Exchange	0
AMM	0
Ocean	0

Figure 22: Interaction 4: Alice wraps DAI into the Ocean

At the end of the sequence, the `BalanceDelta[]` array has only zero values. Hence, there is no need to update Alice's balances in storage.

B.3 Arbitrage Trade via Internal Flash Mint

In this example, Alice will take advantage of an arbitrage opportunity across three different AMMs using a flash mint. Before the initial transaction, the Ocean initializes the `BalanceDelta[]` array (Figure 23).

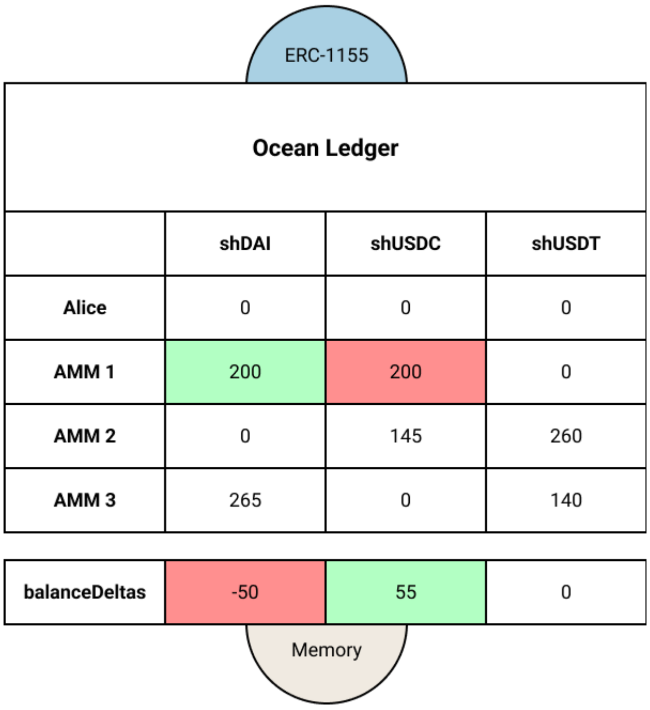

ERC-1155
Ocean Ledger
Memory

	shDAI	shUSDC	shUSDT
Alice	0	0	0
AMM 1	150	255	0
AMM 2	0	145	260
AMM 3	265	0	140
balanceDeltas	0	0	0

Figure 23: Initial state of the ledger before the trade

In the first interaction, Alice swaps shDAI for shUSDC (Figure 24). Because the `BalanceDelta` can go negative, Alice is able to spend shDAI she does not own. Effectively, new shDAI is temporarily minted during the lifecycle of the transaction sequence.

```
Interaction swapDAI {
    bytes32 interactionTypeAndAddress = {ComputeOutputAmount, AMM1};
    uint256 inputToken = shDAI;
    uint256 outputToken = shUSDC;
    uint256 specifiedAmount = 50;
    bytes32 metadata = 0;
}
```



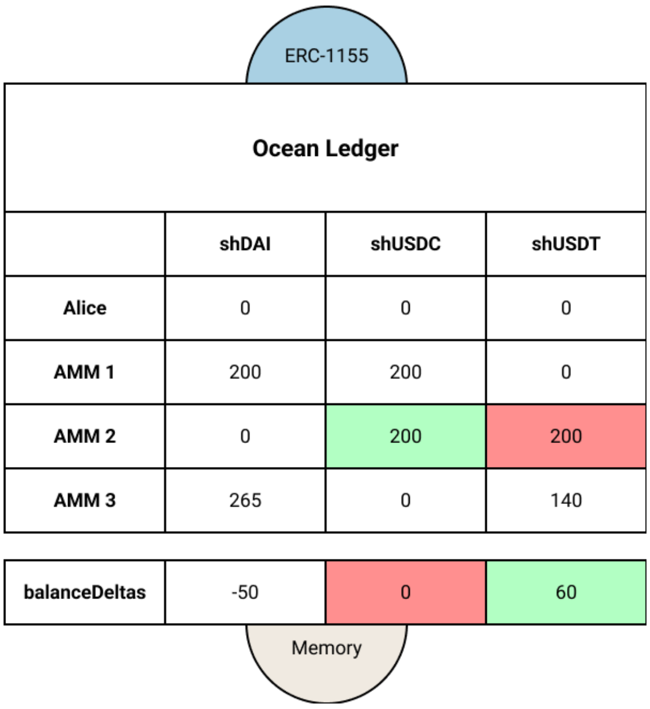
	shDAI	shUSDC	shUSDT
Alice	0	0	0
AMM 1	200	200	0
AMM 2	0	145	260
AMM 3	265	0	140

balanceDeltas	-50	55	0

Figure 24: Interaction 1: Alice swaps shDAI for shUSDC

In the second interaction, Alice swaps shUSDC for shUSDT (Figure 25).

```
Interaction swapUSDC {
    bytes32 interactionTypeAndAddress = {ComputeOutputAmount, AMM2};
    uint256 inputToken = shUSDC;
    uint256 outputToken = shUSDT;
    uint256 specifiedAmount = max;
    bytes32 metadata = 0;
}
```


ERC-1155
Memory

Ocean Ledger

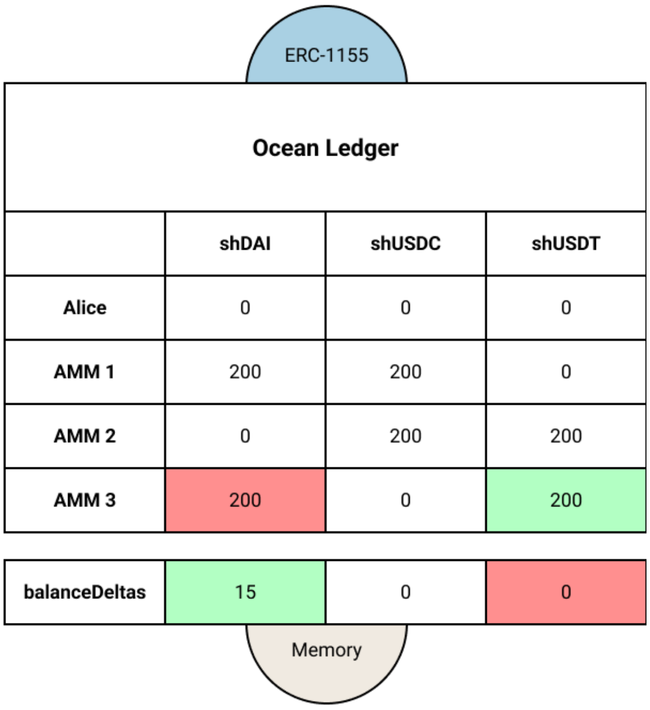
	shDAI	shUSDC	shUSDT
Alice	0	0	0
AMM 1	200	200	0
AMM 2	0	🟩 200	🟥 200
AMM 3	265	0	140

balanceDeltas	-50	🟥 0	🟩 60

Figure 25: Interaction 2: Alice swaps shUSDC for shUSDT

In the third and final interaction, Alice swaps shUSDT for shDAI (Figure 26). Note that Alice now has a positive balance of shDAI in her `BalanceDelta`.

```
Interaction swapUSDC {
    bytes32 interactionTypeAndAddress = {ComputeOutputAmount, AMM2};
    uint256 inputToken = shUSDC;
    uint256 outputToken = shUSDT;
    uint256 specifiedAmount = max;
    bytes32 metadata = 0;
}
```



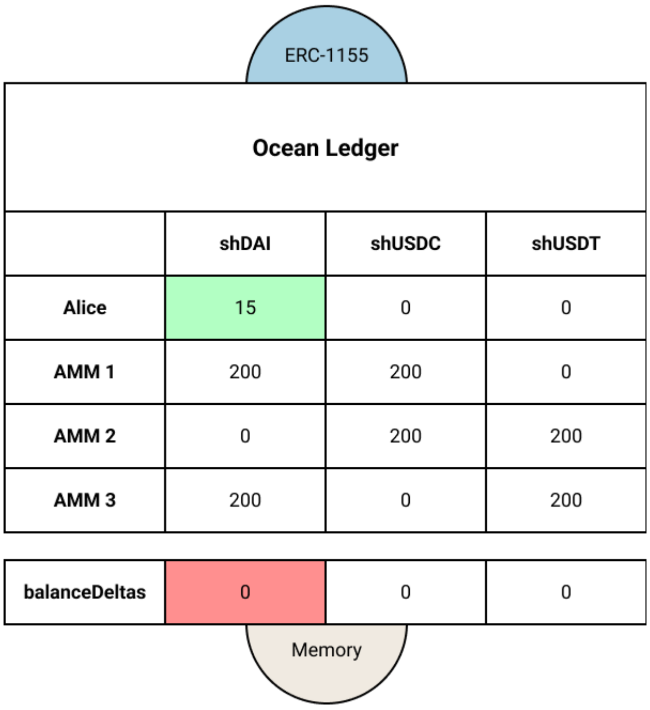
Ocean Ledger

	shDAI	shUSDC	shUSDT
Alice	0	0	0
AMM 1	200	200	0
AMM 2	0	200	200
AMM 3	200	0	200

balanceDeltas	15	0	0

Figure 26: Interaction 3: Alice swaps shUSDT for shDAI

At the end of the sequence, the Ocean increments Alice's balance of shDAI in storage (27).


ERC-1155
Memory

Ocean Ledger

	shDAI	shUSDC	shUSDT
Alice	15	0	0
AMM 1	200	200	0
AMM 2	0	200	200
AMM 3	200	0	200

balanceDeltas	0	0	0

Figure 27: Final state of the ledger after all interactions have been executed