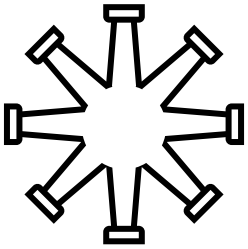
Pipeline & Depot: Dance Through the EVM in 1 Transaction



Publius and Silo Chad
beanstalk.publius@protonmail.com, silochad@bean.farm
evmpipeline.org

Published: November 16, 2022
Modified: November 16, 2022
Whitepaper Version: 1.0.0
Code Version: 1.0.0[1]

"Leave nothing for to-morrow which can be done to-day."

- The Collected Works of Abraham Lincoln edited by Roy P. Basler, Volume II, "Notes for a Law Lecture" (July 1, 1850?), p. 81[2]

Abstract

Using the Ethereum Virtual Machine[3] (EVM) through an Externally-owned account[4] (EOA) is often a painful and time-intensive process. While the censorship resistant and permissionless applications that run on censorship resistant and permissionless computer networks have the potential to dramatically alter the fabric of society, they are less likely to do so if the user experiences they create are worse than centralized competitors. To date, using more than one protocol in the EVM from an EOA frequently requires multiple transactions. A generalized framework to permissionlessly use arbitrary protocols within the EVM, from an EOA, in a single transaction will improve user experiences with decentralized applications. We propose an Ethereum[5]-native pair of contracts that enable this framework for Ether, ERC-20[6], ERC-721[7] and ERC-1155[8] Standard tokens and Beanstalk[9] *Farm* balances and *Silo Deposits*,[10] and can be easily further generalized to include other transferable asset types. A smart contract that can perform an arbitrary series of actions in the EVM in a single transaction creates a sandbox for execution that is loaded and called by a second contract with EIP-712[11] `permit` support for asset approvals. A new function call architecture enables dynamic `callData`.

1 github.com/evmpipeline
2 quod.lib.umich.edu/l/lincoln/lincoln2/1:134?rgn=div1;view=fulltext
3 ethereum.org/en/developers/docs/evm
4 ethereum.org/en/developers/docs/accounts
5 ethereum.org
6 ethereum.org/en/developers/docs/standards/tokens/erc-20
7 ethereum.org/en/developers/docs/standards/tokens/erc-721
8 ethereum.org/en/developers/docs/standards/tokens/erc-1155
9 github.com/BeanstalkFarms/Beanstalk
10 Any italicized terms not defined herein are defined by Beanstalk.
11 eips.ethereum.org/EIPS/eip-712

Table of Contents

1 Introduction

The potential uses of the EVM are infinite. Its composable nature means that infinite functionality need not be implemented by a single protocol. Instead, EVM users can pick and choose the best protocol to use for each action. However, using more than one protocol in a single transaction currently requires a smart contract with a custom function that aggregates the desired protocols uses. Development of custom functions is often expensive and complex.

The need to perform multiple transactions to perform a single desired aggregate action within the EVM creates a horrible user experience. The execution of actions that require multiple transactions typically takes minutes of intermittent user attention and action. Before the execution of subsequent transactions, the reason for attempting to perform the larger action within the EVM could be invalidated by other changes in its state, rendering previous related transaction(s) wasted or worse. There is an additional gas cost associated with each additional transaction used to perform the same aggregate action. While the capabilities of the EVM are incredibly powerful, the likelihood that users take advantage of them in practice is significantly limited by the need to affect single actions in multiple transactions. The high development overhead associated with new functions to implement each combination of other protocols makes creating a good user experience in the EVM excessively difficult.

Pipeline is a standalone contract that creates a sandbox to execute an arbitrary composition of valid actions within the EVM in a single transaction using Ether. *Depot* is a wrapper for *Pipeline* that supports (1) loading Ether and non-Ether assets into *Pipeline*, (2) using them and (3) unloading them from *Pipeline*, in a single transaction. *Clipboard* defines an Advanced Function Call (AFC) architecture that allows subsequent function calls to use the `returnData` of previous ones in a configurable fashion. The combination of *Pipeline*, *Depot* and *Clipboard* allows EVM users to perform arbitrary valid actions, through arbitrarily many protocols, in a single transaction.

2 Previous Work

MakerDAO's `multicall`[12] function allows for arbitrarily many static calls to multiple protocols in a single call.

Uniswap's `multicall`[13] function allows for arbitrarily many function calls to a single protocol in a single transaction.

Beanstalk's `farm`[14] function allows for arbitrarily many function calls to multiple protocols in a single transaction, but requires an upgrade to support each new function call to another protocol.

EIP-712 standardizes a `permit` system to approve the transfer of an amount of an asset as a signed payload in the same transaction that transfers the asset.

[12] github.com/makerdao/multicall
[13] github.com/Uniswap/v3-periphery/blob/main/contracts/base/Multicall.sol
[14] github.com/BeanstalkFarms/Beanstalk/blob/master/protocol/contracts/farm/facets/FarmFacet.sol

3 Pipeline

Pipeline creates a sandbox to execute any series of function calls on any series of protocols through *Pipe* functions. Any assets left in *Pipeline* between transactions can be transferred out by any account. Users *Pipe* a series of *PipeCalls* that each executes a function call to another protocol through *Pipeline*.

3.1 Using Pipeline

In order to safely use *Pipeline*, users must (1) load *Pipeline* with assets, (2) *Pipe* a series of *PipeCalls* to arbitrary protocols and (3) unload all assets from *Pipeline*, in a single transaction.

1. **Load Assets:** *Pipeline* is loaded through any asset transfer function. Users should not load *Pipeline* directly unless only loading Ether. Instead, when loading non-Ether assets, users should prepend an asset transfer function to a *Pipeline* call through *Depot*.
2. ***PipeCalls*:** *Pipeline Pipes* a series of *PipeCalls* to specified protocols executing each *PipeCall* in an isolated environment using the assets loaded in (1).
3. **Unload Assets:** *Pipeline* must be properly unloaded after each use to avoid loss of funds: any leftover assets can be unloaded from *Pipeline* by any account to a destination of their choosing. *Pipeline* can be properly unloaded by both (a) directing the outputs of *PipeCalls* straight to desired destinations and (b) *Piping* asset transfer function calls to transfer assets to desired destinations.

(1) can only be safely performed in the same transaction as (2) and (3) directly through *Pipeline* if the only asset being loaded into it is Ether. In order to safely load non-Ether assets into *Pipeline* users can call *Pipeline* through *Depot*.

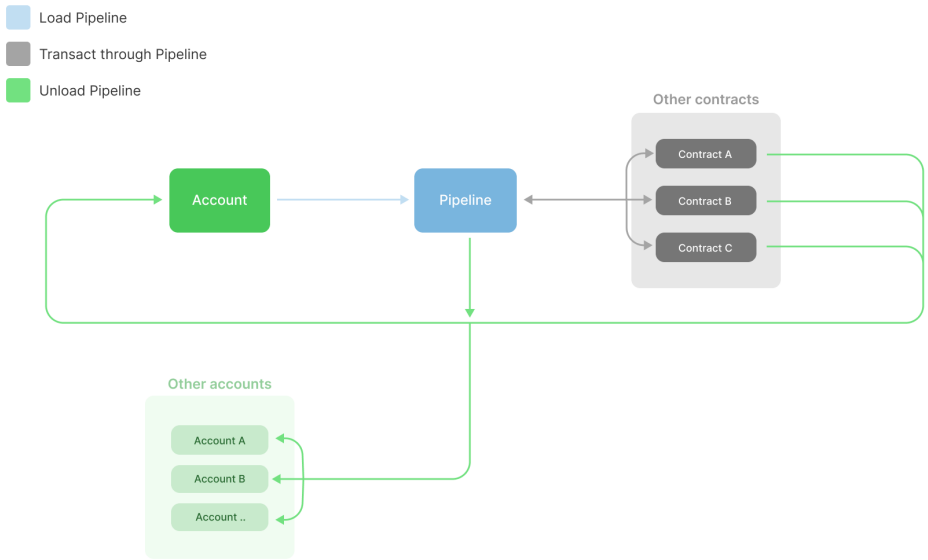


Figure 1: Using Pipeline

3.2 PipeCalls

PipeCalls specify a function call to be executed by *Pipeline*. *Pipeline* supports 2 types of *PipeCalls*: *PipeCall* and *AdvancedPipeCall*.

1. ***PipeCall*:** *PipeCall* makes a function call with a static target address and `callData`.

```
    struct PipeCall {
        address target;
        bytes data;
    }
```

2. ***AdvanccedPipeCall*:** *AdvancedPipeCall* makes a function call with a static target address and both static and dynamic `callData`.

```
    struct AdvancedPipeCall {
        address target;
        bytes data;
        bytes clipboard; // See Section 5 (UPDATE LINK)
    }
```

3.3 Pipe Functions

Pipeline's Pipe functions execute one or more *PipeCalls* or *AdvancedPipeCalls*. *Pipeline* has 3 *Pipe* functions: `pipe`, `multiPipe` and `advancedPipe`

1. `pipe`: `pipe` executes a single *PipeCall*.

```
    function pipe(PipeCall calldata p)
        external
        payable
        returns (bytes memory result);
```

2. `multiPipe`: `multiPipe` executes a list of *PipeCalls*.

```
    function multiPipe(PipeCall[] calldata pipes)
        external
        payable
        returns (bytes[] memory results);
```

3. `advancedPipe`: `advancedPipe` executes a list of *AdvancedPipeCalls*.

```
    function advancedPipe(AdvancedPipeCall[] calldata pipes)
        external
        payable
        returns (bytes[] memory results);
```

4 Depot

Depot wraps *Pipeline's Pipe* functions to facilitate the loading of non-Ether assets in *Pipeline* in the same transaction that loads Ether, *Pipes* calls to other protocols, and unloads *Pipeline*.

Depot has four functions that wrap *Pipeline's Pipe* functions.

1. `pipe`: `pipe` wraps *Pipeline's* `pipe` function but does not support sending Ether.

```
    function pipe(PipeCall calldata p)
        external
        payable
        returns (bytes memory result);
```

2. `etherPipe`: `etherPipe` wraps *Pipeline's* `pipe` function and supports sending Ether.

```
    function etherPipe(PipeCall calldata p, uint256 value)
        external
        payable
        returns (bytes memory result);
```

3. `multiPipe`: `multiPipe` wraps *Pipeline's* `multiPipe` function but does not support sending Ether.

```
    function multiPipe(PipeCall[] calldata pipes)
        external
        payable
        returns (bytes[] memory results);
```

4. `advancedPipe`: `advancedPipe` wraps *Pipeline's* `advancedPipe` function and supports sending Ether.

```
    function advancedPipe(AdvancedPipeCall[] calldata pipes, uint256 value)
        external
        payable
        returns (bytes[] memory results);
```

5 Clipboard

Clipboard allows users to *Copy* return values stored as `returnData` from any *AdvancedPipeCalls* that have already been executed and *Paste* them into the `callData` of the next *AdvancedPipeCall*, in a customizable manner.

Each *AdvancedPipeCalls* includes *Clipboard* to encode (1) how many paste operations to perform, (2) which `returnData` from previous *AdvancedPipeCalls* to *Copy* (*i.e.*, `returnDataIndex`), (2) where to *Copy* from within (3) (*i.e.*, `copyIndex`), and (4) where to *Paste* it in the `callData` of the next *AdvancedPipeCall* (*i.e.*, `pasteIndex`). Bytes are *Pasted* 32 bytes at a time.

Clipboard defines a `Type`, whether to use Ether and how much, and corresponding `pasteParam(s)`, if necessary, for each *AdvancedPipeCall*.

5.1 Type

The first byte of *Clipboard* defines its `Type`, which specifies the amount of *Pasting* operations in the *AdvancedPipeCall*. The `Type` is either `0x00`, `0x01` or `0x02`.

- `0x00` **- Basic function call:** no bytes are *Pasted* (*i.e.*, a static function call).
- `0x01` **- 1** `bytes32` ***Pasting*** **operation:** 1 `bytes32` is *Pasted* from a previous *AdvancedPipeCall's* `returnData`. 1 `pasteParams` are the first 32 bytes of the *Clipboard*.
- `0x02` **- n** `bytes32` ***Pasting*** **operations:** n `bytes32` are *Pasted* from previous *AdvancedPipeCalls'* `returnData`. Bytes 3 through 31 are left empty. Bytes 32 through 32 * (n + 2) are a `bytes32[]`, where each `bytes32` are the `pasteParams` for a given byte *Paste* operation.

5.2 Ether Usage

The second byte of *Clipboard* defines the `Use Ether Flag`, which specifies whether to include Ether in the *AdvancedPipeCall*. The `Use Ether Flag` is either `0x00`, `0x01` or `0x02`.

The *Clipboard* `Use Ether Flag` is defined in the second `byte` of *Clipboard*. It is either `0x00` or `0x01`.

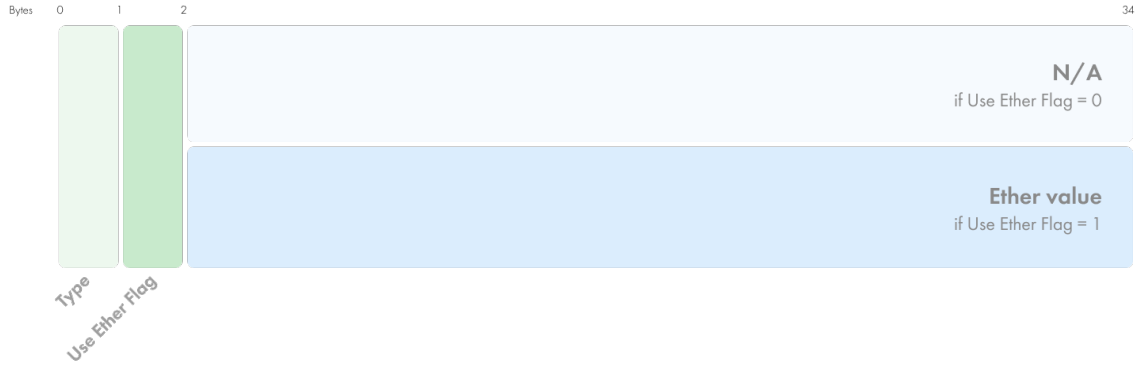
- `0x00` - Do not send Ether in the *AdvancedPipeCall*.
- `0x01` - Send Ether in the *AdvancedPipeCall*. The amount of Ether to send is added as a `uint256` at the end of *Clipboard*.

5.3 pasteParams

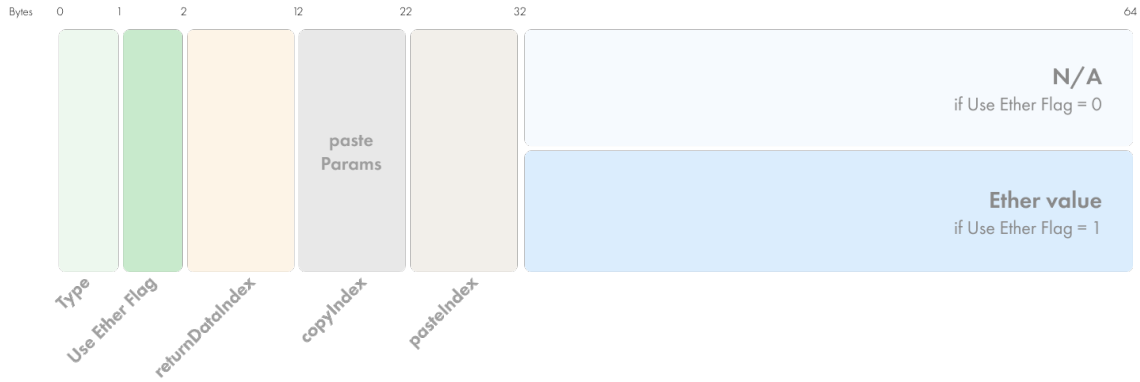
Clipboard `Types` `0x01` and `0x02` accept one or more `pasteParams`, respectively, as inputs that encode where to *Copy* from, what to *Copy*, and where to *Paste* to, 32 bytes per `pasteParams`.

- `Empty` - bytes 0 - 1 are reserved for the Type and `Use Ether Flag`.
- `returnDataIndex` - bytes 2 - 11 specify the index of the `returnData` to *Copy* in the list of `returnData` from previously executed *AdvancedPipeCalls* in the current `advancedPipe` function call, ordered by execution (*e.g.*, `0x0000000000` *Copies* the `returnData` of the 0th *AdvancedPipeCall*, `0x0000000001` *Copies* the `returnData` of the 1st, etc.).
- `copyIndex` - bytes 12 - 21 specify the byte index in the corresponding `returnData[returnDataIndex]` to *Copy*. The first 32 bytes are the length of the return value (*e.g.*, `0x0000000020` *Copies* the 0th return value from `returnData[returnDataIndex]`, `0x0000000040` *Copies* the 1st return value, etc.).
- `pasteIndex` - bytes 22 - 31 specify the byte index of where to *Paste* onto the `callData` of the next *AdvancedPipeCall* (*e.g.*, `0x0000000024` *Pastes* onto the 0th parameter of the next *AdvancedPipeCall's* `callData`, `0x0000000044` *Pastes* onto the 1st parameter, etc.).

Clipboard Type 0x00



Clipboard Type 0x01



Clipboard Type 0x02

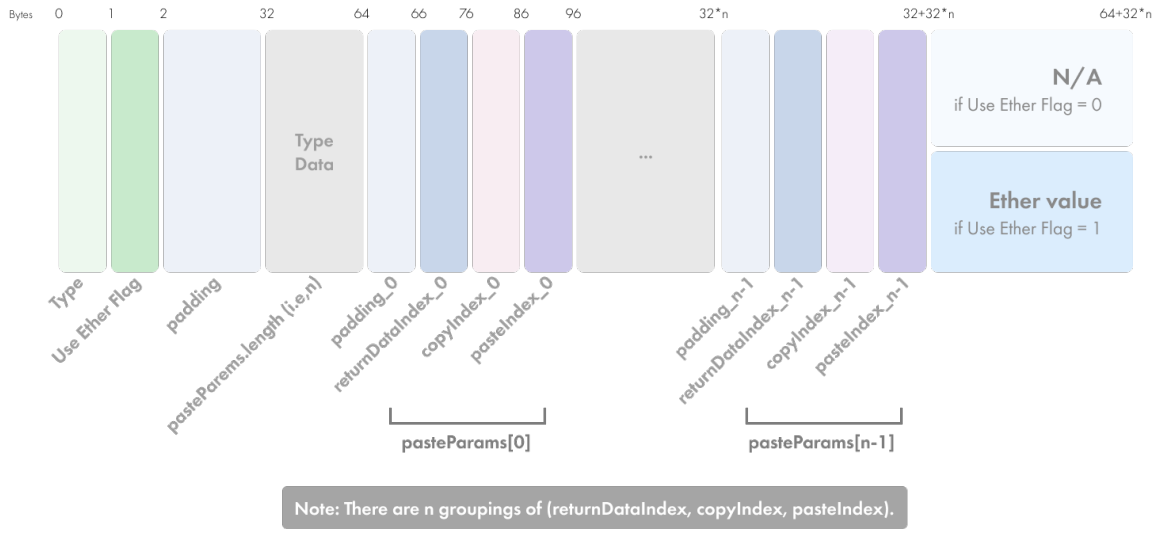


Figure 2: Clipboard

6 Risk

There are numerous risks associated with *Pipeline* and *Depot*. This is not an exhaustive list.

The *Pipeline*, *Depot* and *Clipboard* code bases are novel. Neither has been tested in the "real world" prior to their initial deployment. The open source nature of *Pipeline* and *Depot* means that others can take advantage of any bugs, flaws or deficiencies in them.

The majority of known risks associated with *Pipeline* are from misuse:

- Failure to unload assets from *Pipeline* in the same transaction that loads it will highly likely result in a loss of those assets;
- Approving *Pipeline* to use any assets will highly likely result in a loss of those assets; and
- Improper encoding of *Clipboard* can result in a loss of funds or value. The complexity of properly encoding and decoding *Clipboard* makes verifying correctness difficult.

7 Future Work

Pipeline, *Depot* and *Clipboard* are a work in progress. While *Pipeline* and *Depot* are non-upgradable, they can be easily forked and improved. The following are potential improvements that can be incorporated into them:

- *Depot* easily can be further generalized to include other transferable asset types.
- Currently *Clipboard* only supports *Copying* 32 bytes at a time. Additional *Clipboard* `Types` can be added to allow the entry of an additional `copyLength` parameter.
- Tooling can be improved to make the encoding and decoding of *Clipboard* easier and more accessible.
- The ability to execute *PipeCalls* on various "layers" of the Ethereum network in a single transaction can be implemented.

8 Glossary

The following terms are used throughout this paper:

AdvancedPipeCall - A type of *Pipecall* that makes a function call with a static taget address and both static and dynamic `callData`;

Clipboard - A framework to *Copy* return values stored as `returnData` from any *AdvancedPipeCalls* that have already been executed and *Paste* them into the `callData` of the next *AdvancedPipeCall*, in a customizable manner;

Copy - Add `bytes32 pasteParams` to the *Clipboard*;

Deposit - Assets in the *Silo*;

Depot - A wrapper for *Pipeline* that supports (1) loading Ether and non-Ether assets into *Pipeline*, (2) using them and (3) unloading them from *Pipeline*, in a single transaction;

Farm - Where Beanstalk *Farm* balances are stored;

Paste - Add `bytes32 pasteParams` from the *Clipboard* to `callData` of the next *AdvancedPipeCall* to be executed;

Pipe - *Pipeline's* functions that execute one or more *PipeCalls* or *AdvancedPipeCalls*;

PipeCall - A struct that specifies a function call to be executed by *Pipeline*.

Pipeline - A standalone contract that creates a sandbox to execute an arbitrary composition of valid actions within the EVM in a single transaction using Ether; and

Silo - The Beanstalk DAO.