As we've observed, the Marlowe infrastructure simplifies the use of smart contracts and speeds up development. Marlowe offers various methods to interact with a contract, including deposits, choices, and notifications, which are collectively known as actions. In our specific use case, we'll delve into and analyze the choice action, as it plays a key role in enabling custom and autonomous oracles. Let's begin by understanding the structure of this action, known as Choice.

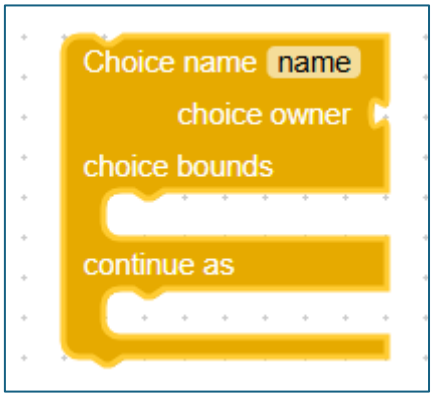

Fig 1. Choice action structure

As shown in **Figure 1**, the choice action structure consists of a name, owner, bounds, and a continuation. Let's explore each of these features in detail.

- Name: This is used to label your action, allowing users to specify which action they want to execute when interacting with a smart contract. It also serves as a reference for future steps in the contract, as the user's choices are stored within the contract and can be accessed in later iterations.
- Owner: This field indicates the ownership or authority to execute the action. It specifies a Party, and whoever holds the associated address or token will have permission to execute the action within that contract.
- Bounds: This field is defined as an interval of integers specified during contract creation. The array of integers represents the options available for the action owner to choose from. For example, if the array contains 0 and 1, these could represent two options: 0 for canceling the contract and 1 for proceeding with it. This feature provides significant flexibility for the contract creator, allowing them to define different execution paths within the contract.
- Continuation: Here, we can specify which contract will follow once the choice action has been successfully executed.

At this stage, you might be asking where to specify whether to use a custom or autonomous oracle, and how oracles are related to a choice action. To clarify, an oracle is a mechanism that provides external information—data not inherently available on the

blockchain—into a smart contract. This external information is input into the contract through the choice bounds, and the entity responsible for setting these bounds is the owner of the choice, which we refer to as the oracle.

Now, let's address the difference between a custom oracle and an autonomous oracle. A custom oracle is what we previously encountered in the examples from Milestone 3, where we specify an address to represent a role within our contract, such as an auditor. In this case, we can designate trusted parties who can make a choice within our contract when prompted. It’s not considered autonomous because the owner of that address is a person who periodically enters the contract, checks for any available actions, and then executes them. This reliance on human intervention is why we refer to it as non-autonomous.

Let's clarify what an autonomous oracle is. A straightforward answer might suggest that it's something that doesn't rely on human intervention—like a bot that holds the authorization keys, accesses external information, follows an algorithm, and then executes actions. While this is true for enterprises that hold exclusive information or use private APIs, there's an important aspect missing for broader blockchain applications: decentralization and trustlessness. For an autonomous oracle, what we aim for is a mechanism to bring external information onto the blockchain in a decentralized and trustless manner, without requiring human intervention in direct way.

Now that we understand what a Marlowe choice action is, what an oracle is, and how they are connected, the next question is whether any autonomous oracles are already available in the Cardano ecosystem. The answer is yes. There are commercial examples like Charli3, as well as case studies and research conducted by IOG, Wolfram, Marlowe, and TxPipe. Charli3 have developed oracles that integrate directly with smart contracts using Plutus syntax. Since Marlowe is an abstraction of Plutus, some infrastructure and code are needed to bridge the gap, enabling these commercial oracles to interact with Marlowe contracts, specifically through the choice input syntax.

So, we have now identified the focus of our analysis. We'll examine what each of these companies has developed, how they connect with each other, their architecture, and the trade-offs involved. Our study will be divided into cases, with each case discussing a specific solution architecture. This architecture may involve a combination of commercial oracles or research organizations exploring the best ways to integrate oracles with Marlowe contracts or improve Oracles systems.

Case #1: Charlie Oracles

Charli3 has recently released an MVP of a new architecture designed to enhance oracles by providing real-time price feeds on demand. To understand this architecture in context, we can categorize it into two main types: pull-based and push-based oracles. Protocols on Cardano can now use Charli3 to request data as needed (pull-based) rather than receiving it at set intervals (push-based). This approach offers two key benefits: a) reduced costs, as data is only pulled when required, and b) the ability to integrate payment for data delivery at the time of user request. While the latter offers greater flexibility, it does involve more integration work, but it eliminates the need to continuously fund feeds with tokens or payments.

Figure 2 illustrates the architecture of the push-based oracle, which consists of node pools and smart contracts, and highlights the need for continuous funding of a UTxO. Data is inserted into the blockchain through a Feed UTxO, which is accessed via its inline datum. It's important to note that while a regular transaction or smart contract can directly utilize the data from this UTxO, additional steps are required for Marlowe smart contracts to translate this datum into a choice action. This will be discussed in future examples, with Case #1 focusing on a commercial smart contract architecture.

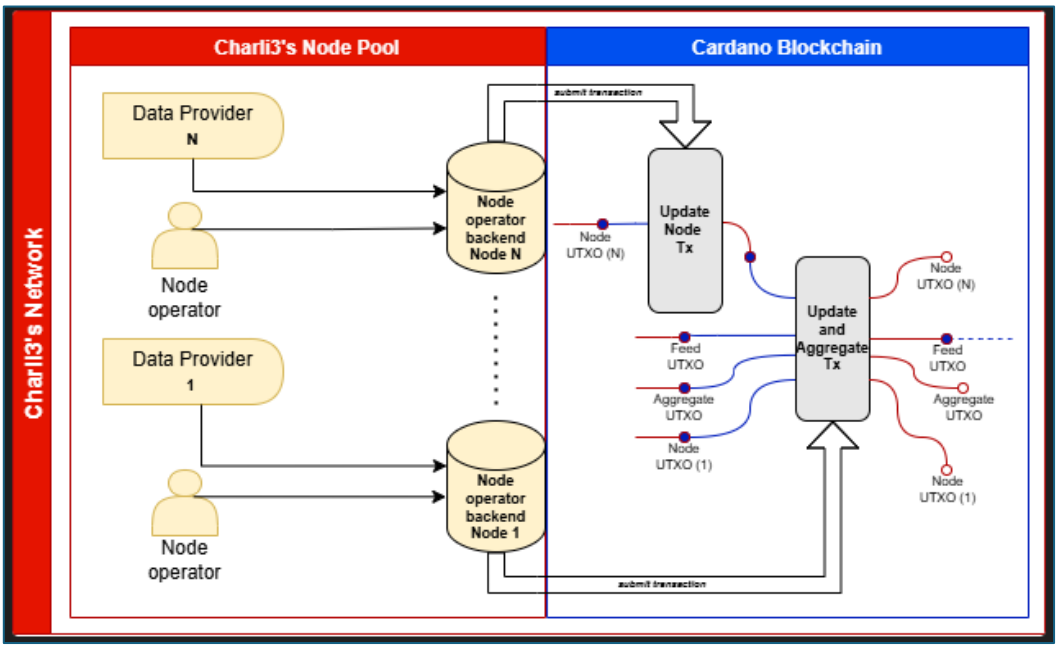

Fig 2. Node Operators and smart contracts

The architectural difference for pull-based oracles is minimal, with the key distinction being how the node operator manages and updates their UTxO. In push-based oracles, the update frequency is predefined, whereas in pull-based oracles, it is determined by user requests. Additional tooling is incorporated into this infrastructure to monitor the chain for new events.

Case #2: IOG + Tx-Pipe + Charlie3 Research related to Decentralized Oracle Integration

Now that we understand how decentralized oracles function and how they convert off-chain information into on-chain data, we will explore a potential method to integrate this data into actionable steps within Marlowe contracts. As explained in the last section, in Marlowe, a Choice action is used to request outside data, and using a specific combination of name and owner it can be stated that the data should be coming from an Oracle. The method that we will analyze is called Marlowe Oracle service (MOS).

The Marlowe Oracle Service (MOS) is an active service designed to query the blockchain for Marlowe contracts that require data feeds. It can filter known feeds to construct, balance, and submit transactions that supply these feeds. The MOS delivers feeds to two types of choice parties: Address and Role. The MOS address will be publicly accessible, allowing any choice associated with that address to be resolved via a regular transaction that completes the choice. For parties identified by a publicly known Role, the transaction will also involve consuming a UTxO containing a validator known as the Oracle Bridge Validator. Therefore, the system comprises two main components: an off-chain backend responsible for scanning and transaction construction, and the Oracle Bridge Validator.

<u>Off-chain backend:</u>

The backend utilizes the Marlowe Runtime to scan active contracts and then filters them based on the following criteria:

- It has a choice action available
- The choice name is a valid feed
- Owner address is the service address or role token

The diagram illustrating how this function for an owner designated as an address is shown in **Figure 3**. This is used for private Oracles.

The Marlowe Oracle Service queries the Marlowe Runtime for all contracts (1), filters for those requiring Oracle input (2), and identifies the appropriate data source (3). It then sends the feed and contract to the Runtime to apply the Choice input and generates an unsigned transaction (4), which is subsequently signed (5) and submitted (6).

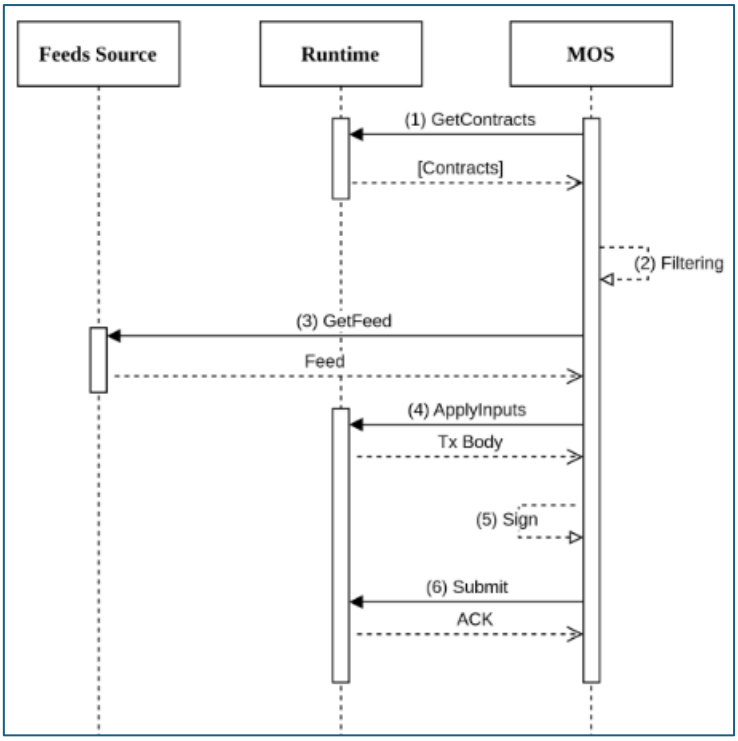

Fig 3. Owner as Address. MOS architecture

The diagram illustrating how this function for an owner designated as a role is shown in **Figure 4**. This is used for contracts that want to utilize decentralized oracles like Charlie3 which is our case.

The Marlowe Oracle Service queries the Marlowe Runtime for contracts (1), filters for those needing Oracle input (2), and then queries the blockchain via the Chain Indexer for UTxOs of the Oracle Bridge validator (3). It filters these UTxOs to find the one with the corresponding Oracle role token (4) and then queries the Chain Indexer again for the UTxO containing the requested Oracle Feed (5). The service sends this value and the contract to the Runtime to apply the Choice input and generate the unsigned transaction body (6). The MOS adds the Bridge Validator UTxO as an input and the Oracle Feed UTxO as a reference input (7). Finally, the transaction is signed (8) and submitted (9).

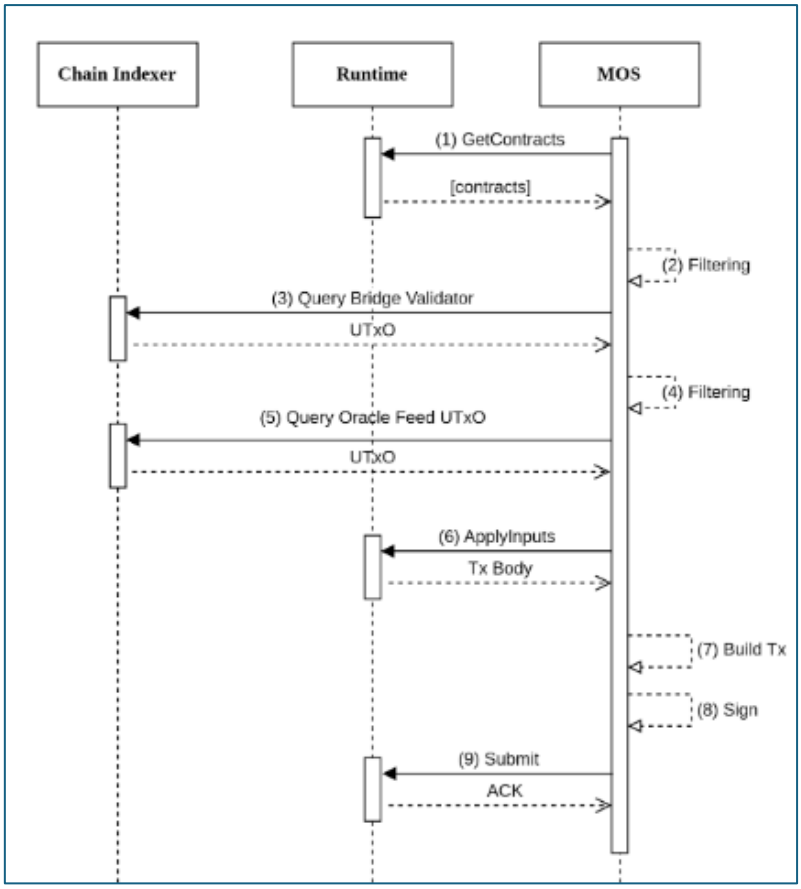

Fig 4. Owner as Role Token. MOS architecture

Oracle Bridge Validator:

The bridge validator is responsible for ensuring the effective use of decentralized oracles when required by a Marlowe contract. It will have the following parameters:

- The hash of the Marlowe validator
- The currency symbol of the token in the oracle's reference UTxO
- The token name of the token in the oracle's reference UTxO
- The name of the Marlowe Choice where the contract will receive oracle input

For each Marlowe contract that intends to use the bridge validator, a new UTxO at that address must be created. This UTxO will contain the role token and include a reference (in the datum) to the thread token required by the Marlowe contract.

Figure 5 illustrates an example of a transaction that must be created to meet the Oracle bridge validator's requirements.

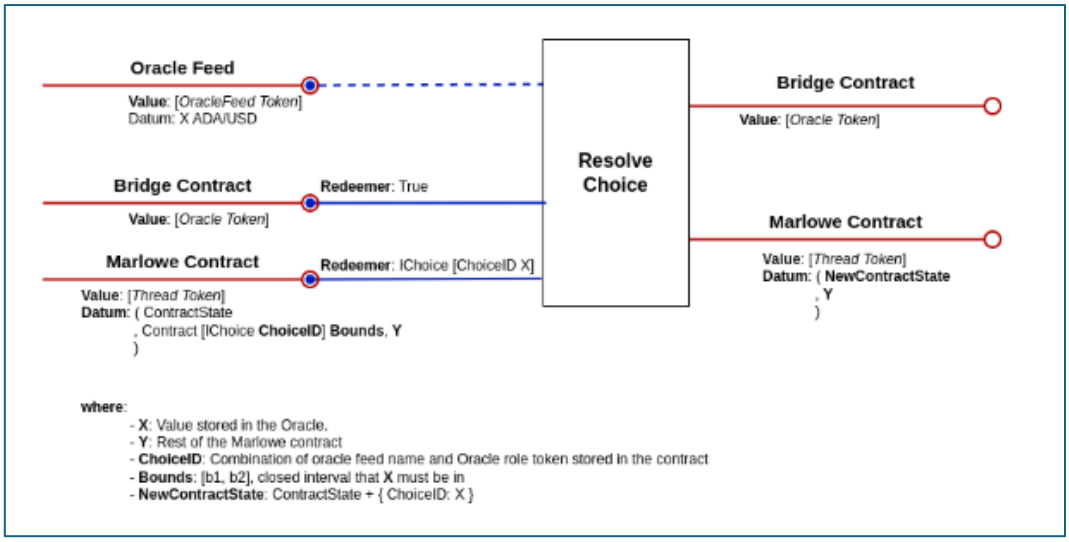

Fig 5. Specification of a transaction using a role token and bridge validator

This first method provides a clear explanation of the components involved in the solution. In the subsequent methods, our focus will shift to identifying architectural changes and evaluating which performs better based on a set of parameters. For this method, I have identified the following key aspects

- Chain watch mechanism and filtering of Marlowe Contracts: The engine used here is the Marlowe Runtime itself, where a search is conducted with indexing limited to active contracts. This could present a challenge depending on the number of active contracts during the search. Furthermore, once the active contracts are identified, they must be filtered one at a time to find those requiring a choice input and matching the correct name. These concerns may impact the efficiency of the solution. However, it's important to note that this is more about optimizing the runtime rather than the MOS architecture.
- Chain Indexer: The effectiveness of this will depend on the indexing implemented. In this case, we observe that indexing and filtering are utilized in this solution.
- Aiken on-chain validators: In this approach, we assess the number of steps/transactions required before validation. Each contract must create a custom UTxO within its specific bridge validator, which entails multiple transaction steps, associated costs, and coordination efforts.
- Real-time price feeds: Charlie3 was mentioned.
- Financial Logic: Marlowe Smart contracts which are constraints within Marlowe Semantics.
- Financial Sustainability: It was not described how compensation will work for those running the MOS.
- Request method: Pull-based Request. This is more efficient than push-based request where updates need to be done at a certain frequency independent of users' needs.

Case #3: Wolfram Marlowe Smart Contract Execution

It is important to note that this solution is still under development by the Wolfram team, and they are currently at milestone #1 out of five. We will review the available open-source components and compare them with other architectures.

In this instance, the Wolfram team notes that three essential elements are required for a Marlowe application to be ready for live deployment:

- Financial Logic
- Real-time price feeds
- Autonomous execution

We can observe macro components akin to the key structures identified in Case #2.

The general architecture is described in **Figure 6**:

<u>On-Chain:</u>

- Wolfram Oracle Contract

<u>Off-chain:</u>

- Wolfram Oracle Harvester
- Wolfram Price Feed Infrastructure

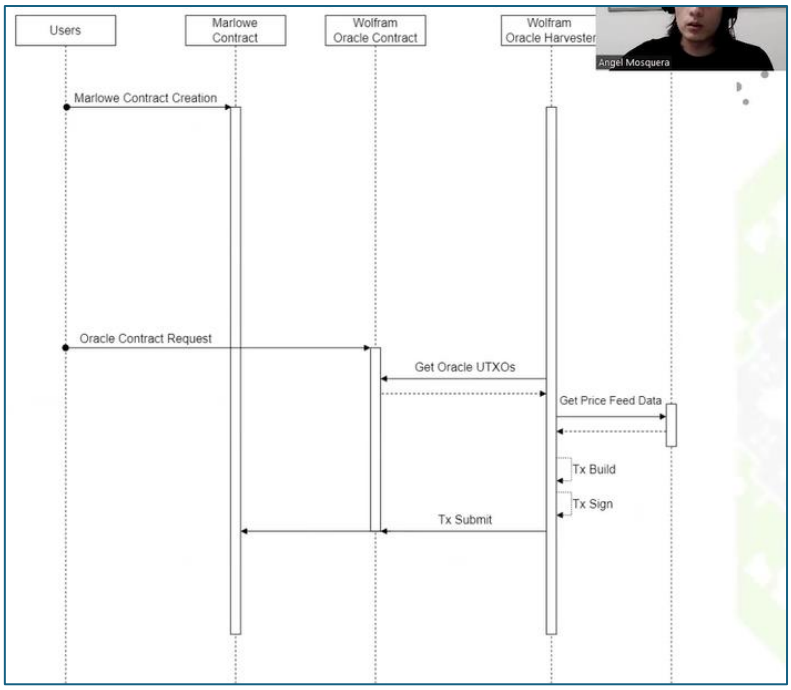

Fig 6. General Wolfram Architecture for running smart contracts.

In their architecture, they have not yet detailed the complete structure of the price feed platform. However, it is evident that it is defined by an address rather than a UTxO. This means that when a user wants to retrieve real-time data from the Oracle, a payment to the Oracle is required to a specific address, with specific parameters provided in the datum, after which the Oracle responds. This mechanism is illustrated in **Figure 7**.

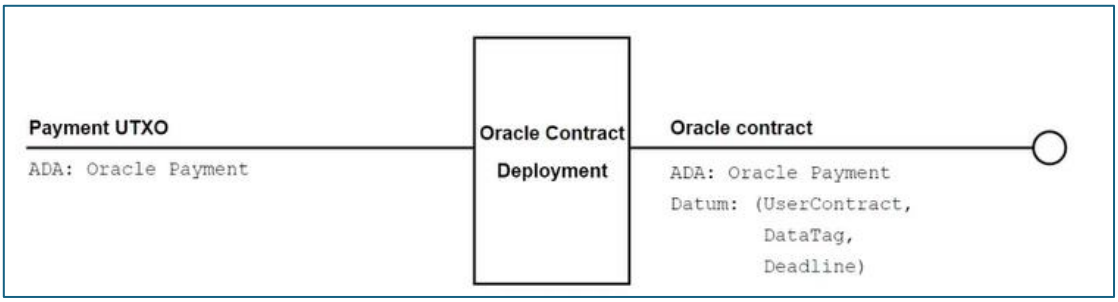

Fig 7. Oracle Contract implementation

Once the constraints are satisfied, the oracle claims the ADA inside the UTxO as payment for the service, **Figure 8**.

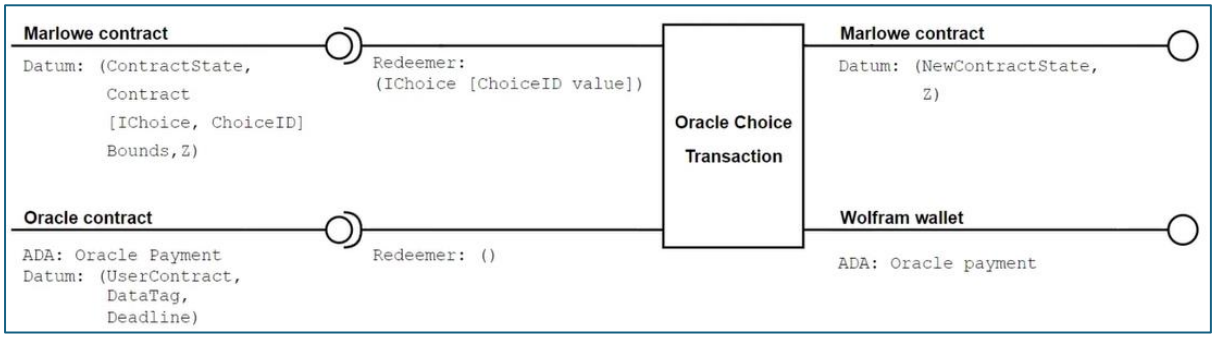

Fig 8. Oracle Choice Transaction

Based on the information at hand, let's evaluate this architecture using the same key aspects outlined in Case #2:

- Chain watch mechanism and filtering of Marlowe Contracts: This component of the system in Case #3 is referred to as the Wolfram Oracle Harvester. While they have not yet detailed how it will be constructed, as mentioned earlier, it is more of a feature of the runtime itself. Therefore, any improvements to the runtime will be reflected across all cases.
- Chain Indexer: They have not yet provided the details of the actual architecture. However, since calling an Oracle is based on an address rather than a UTxO, one

less step is required. Nonetheless, this still needs to be analyzed, as despite having one fewer step, we don't yet know the architecture of the price feed or whether it is decentralized.

- Aiken on-chain validators: The validator structure is simpler than that in Case #2; they have already incorporated a price check that includes the Oracle payment. However, their analysis does not account for the thread token of a Marlowe contract. While the structure is more straightforward, we will need to wait for a formal audit to verify the security and decentralization of the architecture.
- Real-time price feeds: Wolfram is not utilizing Charlie3; instead, they have developed a custom real-time price feed called the Wolfram Price Feed Infrastructure. This project is being developed in parallel and falls outside the scope of this analysis. Ultimately, when it comes to real-time price feeds, the key factors are accuracy, security, and decentralization. It's possible they aim to create an alternative that could compete with Charlie3 or Orcfax.
- Financial Logic: Marlowe Smart contracts which are constraints within Marlowe Semantics.
- Financial Sustainability: This architecture includes its financial model, and the features embedded within a smart contract. In Case #2, these elements were not included and remain undefined.
- Request method: Pull-based Request. This is more efficient than push-based request where updates need to be done at a certain frequency independent of users' needs.

In conclusion, when comparing the architectures from Case #2 and Case #3, the primary differences lie in on-chain validation and oracle selection. Tx-Pipe emphasizes utilizing existing, proven technology to build tools on top of, while Wolfram is developing custom software for price feeds, which still needs to be validated, tested, and audited.

At this stage, for a real-world project, I would opt for the Tx-Pipe architecture as it is more robust and tested. The only significant aspect missing from the Tx-Pipe + IOG architecture is a clear monetization strategy for the MOS, but I believe this could be easily implemented or operated as a private service.

Case #4: Open Marlowe Oracle Protocol

This initiative, though still in the planning stages, represents a more holistic approach compared to other models. It involves significant changes to both the Marlowe validator and the runtime. Unlike previous models, this protocol does not rely on off-chain components to search for contracts requiring input. Instead, it focuses on developing modular API calls that request specific data and include payment mechanisms. This marks a significant paradigm shift, encompassing more comprehensive changes than other models. While I believe this approach will become the preferred method in the future once approved, for now, the recommended structure for commercial Dapps is the TX-pipe + IOG model.

Marlowe Oracle Protocol Improvements and Specification:

Rationale: Marlowe introduces a simple yet powerful approach to Oracle integration through a novel request-response-based model, providing a flexible and secure API for Oracle service fee payments. This approach offers several advantages, including simplicity, flexibility, and potential for parallelization, making it broadly applicable across the ecosystem.

Outputs:

Validator changes:

- Introduce a script to guard the output Marlowe state after contract finalization, locking the Oracle data (Marlowe choices) and thread token. The results will be consumable only by a predefined NFT holder, with future potential for inline datum publishing.
- Enable the output of part of the state (choices) to the Marlowe validator.

Runtime changes:

- Add support for the new state-guarding script.
- Integrate the validator changes and offer an option in the REST API to enable state output.