---

title: Oracles description: Oracles provide Ethereum smart contracts with access to real-world data, unlocking more use-cases and greater value for users. lang: en

---

Oracles are data feeds that make off-chain data sources available to the blockchain for smart contracts. This is necessary because Ethereum-based smart contracts cannot, by default, access information stored outside the blockchain network.

Giving smart contracts the ability to execute using off-chain data extends the utility and value of decentralized applications. For instance, on-chain prediction markets rely on oracles to provide information about outcomes that they use to validate user predictions. Suppose Alice bets 20 ETH on who will become the next U.S. President. In that case, the prediction-market dapp needs an oracle to confirm election results and determine if Alice is eligible for a payout.

Prerequisites {#prerequisites}

This page assumes the reader is familiar with Ethereum fundamentals, includingnodes, consensus mechanisms, and the EVM. You should also have a good grasp ofsmart contracts and smart contract anatomy, especially events.

What is a blockchain oracle? {#what-is-a-blockchain-oracle}

Oracles are applications that source, verify, and transmit external information (i.e. information stored off-chain) to smart contracts running on the blockchain. Besides “pulling” off-chain data and broadcasting it on Ethereum, oracles can also “push” information from the blockchain to external systems, e.g., unlocking a smart lock once the user sends a fee via an Ethereum transaction.

Without an oracle, a smart contract would be limited entirely to on-chain data.

Oracles differ based on the source of data (one or multiple sources), trust models (centralized or decentralized), and system architecture (immediate-read, publish-subscribe, and request-response). We can also distinguish between oracles based on whether they retrieve external data for use by on-chain contracts (input oracles), send information from the blockchain to the off-chain applications (output oracles), or perform computational tasks off-chain (computational oracles).

Why do smart contracts need oracles? {#why-do-smart-contracts-need-oracles}

Many developers see smart contracts as code running at specific addresses on the blockchain. However, a moregeneral view of smart contracts is that they are self-executing software programs capable of enforcing agreements between parties once specific conditions are met - hence the term “smart contracts.”

But using smart contracts to enforce agreements between people isn't straightforward, given that Ethereum is deterministic. A deterministic system is one that always produces the same results given an initial state and a particular input, meaning there is no randomness or variation in the process of computing outputs from inputs.

To achieve deterministic execution, blockchains limit nodes to reaching consensus on simple binary (true/false) questions using *only* data stored on the blockchain itself. Examples of such questions include:

- “Did the account owner (identified by a public key) sign this transaction with the paired private key?”
- “Does this account have enough funds to cover the transaction?”
- “Is this transaction valid in the context of this smart contract?”, etc.

If blockchains received information from external sources (i.e. from the real world), determinism would be impossible to achieve, preventing nodes from agreeing on the validity of changes to the blockchain’s state. Take for example a smart contract that executes a transaction based on the current ETH-USD exchange rate obtained from a traditional price API. This figure is likely to change frequently (not to mention that the API could get deprecated or hacked), meaning nodes executing the same contract code would arrive at different results.

For a public blockchain like Ethereum, with thousands of nodes around the world processing transactions, determinism is

critical. With no central authority serving as a source of truth, nodes need mechanisms for arriving at the same state after applying the same transactions. A case whereby node A executes a smart contract's code and gets "3" as a result, while node B gets "7" after running the same transaction would cause consensus to break down and eliminate Ethereum's value as a decentralized computing platform.

This scenario also highlights the problem with designing blockchains to pull information from external sources. Oracles, however, solve this problem by taking information from off-chain sources and storing it on the blockchain for smart contracts to consume. Since information stored on-chain is unalterable and publicly available, Ethereum nodes can safely use the oracle imported off-chain data to compute state changes without breaking consensus.

To do this, an oracle is typically made up of a smart contract running on-chain and some off-chain components. The on-chain contract receives requests for data from other smart contracts, which it passes to the off-chain component (called an oracle node). This oracle node can query data sources—using application programming interfaces (APIs), for example—and send transactions to store the requested data in the smart contract's storage.

Essentially, a blockchain oracle bridges the information gap between the blockchain and the external environment, creating "hybrid smart contracts". A hybrid smart contract is one that functions based on a combination of on-chain contract code and off-chain infrastructure. Decentralized prediction markets are an excellent example of hybrid smart contracts. Other examples might include crop insurance smart contracts that pay out when a set of oracles determine that certain weather phenomena have taken place.

What is the oracle problem? {#the-oracle-problem}

Oracles solve an important problem, but also introduce some complications, e.g.:

- How do we verify that the injected information was extracted from the correct source or hasn't been tampered with?
- How do we ensure that this data is always available and updated regularly?

The so-called "oracle problem" demonstrates the issues that come with using blockchain oracles to send inputs to smart contracts. Data from an oracle must be correct for a smart contract to execute correctly. Further, having to 'trust' oracle operators to provide accurate information undermines the 'trustless' aspect of smart contracts.

Different oracles offer different solutions to the oracle problem, which we explore later. Oracles are typically evaluated on how well they can handle the following challenges:

1. **Correctness**: An oracle should not cause smart contracts to trigger state changes based on invalid off-chain data. An oracle must guarantee *authenticity* and *integrity* of data. Authenticity means the data was gotten from the correct source, while integrity means the data remained intact (i.e. wasn't altered) before being sent on-chain.
2. **Availability**: An oracle should not delay or prevent smart contracts from executing actions and triggering state changes. This means that data from an oracle must be *available on request* without interruption.
3. **Incentive compatibility**: An oracle should incentivize off-chain data providers to submit correct information to smart contracts. Incentive compatibility involves *attributability* and *accountability*. Attributability allows for linking a piece of external information to its provider, while accountability bonds data providers to the information they give, so they can be rewarded or penalized based on the quality of information provided.

How does a blockchain oracle service work? {#how-does-a-blockchain-oracle-service-work}

Users {#users}

Users are entities (i.e., smart contracts) that need information external to the blockchain to complete specific actions. The basic workflow of an oracle service starts with the user sending a data request to the oracle contract. Data requests will usually answer some or all of the following questions:

1. What sources can off-chain nodes consult for the requested information?
2. How do reporters process information from data sources and extract useful data points?
3. How many oracle nodes can participate in retrieving the data?
4. How should discrepancies in oracle reports be managed?
5. What method should be implemented in filtering submissions and aggregating reports into a single value?

Oracle contract {#oracle-contract}

The oracle contract is the on-chain component for the oracle service. It listens for data requests from other contracts, relays data queries to oracle nodes, and broadcasts returned data to client contracts. This contract may also perform some computation on the returned data points to produce an aggregate value to send to the requesting contract.

The oracle contract exposes some functions which client contracts call when making a data request. Upon receiving a new query, the smart contract will emit a log event with details of the data request. This notifies off-chain nodes subscribed to the log (usually using something like the JSON-RPC `eth_subscribe` command), who proceed to retrieve data defined in the log event.

Below is an example oracle contract by Pedro Costa. This is a simple oracle service that can query off-chain APIs upon request by other smart contracts and store the requested information on the blockchain:

```solidity pragma solidity >=0.4.21 <0.6.0;

contract Oracle { Request[] requests; //list of requests made to the contract uint currentId = 0; //increasing request id uint minQuorum = 2; //minimum number of responses to receive before declaring final result uint totalOracleCount = 3; // Hardcoded oracle count

// defines a general api request struct Request { uint id; //request id string urlToQuery; //API url string attributeToFetch; //json attribute (key) to retrieve in the response string agreedValue; //value from key mapping(uint => string) answers; //answers provided by the oracles mapping(address => uint) quorum; //oracles which will query the answer (1=oracle hasn't voted, 2=oracle has voted) }

//event that triggers oracle outside of the blockchain event NewRequest ( uint id, string urlToQuery, string attributeToFetch );

//triggered when there's a consensus on the final result event UpdatedRequest ( uint id, string urlToQuery, string attributeToFetch, string agreedValue );

function createRequest ( string memory _urlToQuery, string memory _attributeToFetch ) public { uint length = requests.push(Request(currentId, _urlToQuery, _attributeToFetch, "")); Request storage r = requests[length-1];

```
// Hardcoded oracles address
r.quorum[address(0x6c2339b46F41a06f09CA0051ddAD54D1e582bA77)] = 1;
r.quorum[address(0xb5346CF224c02186606e5f89EACC21eC25398077)] = 1;
r.quorum[address(0xa2997F1CA363D11a0a35bB1Ac0Ff7849bc13e914)] = 1;

// launch an event to be detected by oracle outside of blockchain
emit NewRequest (
  currentId,
  _urlToQuery,
  _attributeToFetch
);

// increase request id
currentId++;
```

}

//called by the oracle to record its answer function updateRequest ( uint _id, string memory _valueRetrieved ) public {

```
Request storage currRequest = requests[_id];

//check if oracle is in the list of trusted oracles
//and if the oracle hasn't voted yet
```
```

```
if(currRequest.quorum[address(msg.sender)] == 1){

  //marking that this address has voted
  currRequest.quorum[msg.sender] = 2;

  //iterate through "array" of answers until a position if free and save the retrieved value
  uint tmpI = 0;
  bool found = false;
  while(!found) {
    //find first empty slot
    if(bytes(currRequest.answers[tmpI]).length == 0){
      found = true;
      currRequest.answers[tmpI] = _valueRetrieved;
    }
    tmpI++;
  }

  uint currentQuorum = 0;

  //iterate through oracle list and check if enough oracles(minimum quorum)
  //have voted the same answer has the current one
  for(uint i = 0; i < totalOracleCount; i++){
    bytes memory a = bytes(currRequest.answers[i]);
    bytes memory b = bytes(_valueRetrieved);

    if(keccak256(a) == keccak256(b)){
      currentQuorum++;
      if(currentQuorum >= minQuorum){
        currRequest.agreedValue = _valueRetrieved;
        emit UpdatedRequest (
          currRequest.id,
          currRequest.urlToQuery,
          currRequest.attributeToFetch,
          currRequest.agreedValue
        );
      }
    }
  }
}

}}```
```

Oracle nodes {#oracle-nodes}

The oracle node is the off-chain component of the oracle service. It extracts information from external sources, such as APIs hosted on third-party servers, and puts it on-chain for consumption by smart contracts. Oracle nodes listen for events from the on-chain oracle contract and proceed to complete the task described in the log.

A common task for oracle nodes is sending a HTTP GET request to an API service, parsing the response to extract relevant data, formatting into a blockchain-readable output, and sending it on-chain by including it in a transaction to the oracle contract. The oracle node may also be required to attest to the validity and integrity of submitted information using “authenticity proofs”, which we explore later.

Computational oracles also rely on off-chain nodes to perform computational tasks that would be impractical to execute on-chain, given gas costs and block size limits. For example, the oracle node may be tasked with generating a verifiably random figure (e.g., for blockchain-based games).

Oracle design patterns {#oracle-design-patterns}

Oracles come in different types, including *immediate-read*, *publish-subscribe*, and *request-response*, with the latter two being the most popular among Ethereum smart contracts. Here we briefly describe the publish-subscribe and request-response models.

Publish-subscribe oracles {#publish-subscribe-oracles}

This type of oracle exposes a “data feed” which other contracts can regularly read for information. The data in this case is expected to change frequently, so client contracts must listen for updates to the data in the oracle’s storage. An example is an oracle that provides the latest ETH-USD price information to users.

Request-response oracles {#request-response-oracles}

A request-response setup allows the client contract to request arbitrary data other than that provided by a publish-subscribe oracle. Request-response oracles are ideal when the dataset is too large to be stored in a smart contract's storage, and/or users will only need a small part of the data at any point in time.

Although more complex than publish-subscribe models, request-response oracles are basically what we described in the previous section. The oracle will have an on-chain component that receives a data request and passes it to an off-chain node for processing.

Users initiating data queries must cover the cost of retrieving information from the off-chain source. The client contract must also provide funds to cover gas costs incurred by the oracle contract in returning the response via the callback function specified in the request.

Centralized vs. decentralized oracles {#types-of-oracles}

Centralized oracles {#centralized-oracles}

A centralized oracle is controlled by a single entity responsible for aggregating off-chain information and updating the oracle contract's data as requested. Centralized oracles are efficient since they rely on a single source of truth. They may function better in cases where proprietary datasets are published directly by the owner with a widely accepted signature. However, they bring downsides as well:

Low correctness guarantees {#low-correctness-guarantees}

With centralized oracles, there's no way to confirm if the information provided is correct or not. Even "reputable" providers can go rogue or get hacked. If the oracle becomes corrupt, smart contracts will execute based on bad data.

Poor availability {#poor-availability}

Centralized oracles aren't guaranteed to always make off-chain data available to other smart contracts. If the provider decides to turn off the service or a hacker hijacks the oracle's off-chain component, your smart contract is at risk of a denial of service (DoS) attack.

Poor incentive compatibility {#poor-incentive-compatibility}

Centralized oracles often have poorly designed or non-existent incentives for the data provider to send accurate/unaltered information. Paying an oracle for for correctness does not guarantee honesty. This problem gets bigger as the amount of value controlled by smart contracts increases.

Decentralized oracles {#decentralized-oracles}

Decentralized oracles are designed to overcome the limitations of centralized oracles by eliminating single points of failure. A decentralized oracle service comprises multiple participants in a peer-to-peer network that form consensus on off-chain data before sending it to a smart contract.

A decentralized oracle should (ideally) be permissionless, trustless, and free from administration by a central party; in reality, decentralization among oracles is on a spectrum. There are semi-decentralized oracle networks where anyone can participate, but with an "owner" that approves and removes nodes based on historical performance. Fully decentralized oracle networks also exist: these usually run as standalone blockchains and have defined consensus mechanisms for coordinating nodes and punishing misbehavior.

Using decentralized oracles comes with the following benefits:

High correctness guarantees {#high-correctness-guarantees}

Decentralized oracles attempt to achieve correctness of data using different approaches. This includes using proofs

attesting to the authenticity and integrity of the returned information and requiring multiple entities to collectively agree on the validity of off-chain data.

Authenticity proofs {#authenticity-proofs}

Authenticity proofs are cryptographic mechanisms that enable independent verification of information retrieved from external sources. These proofs can validate the source of the information and detect possible alterations to the data after retrieval.

Examples of authenticity proofs include:

Transport Layer Security (TLS) proofs: Oracle nodes often retrieve data from external sources using a secure HTTP connection based on the Transport Layer Security (TLS) protocol. Some decentralized oracles use authenticity proofs to verify TLS sessions (i.e., confirm the exchange of information between a node and a specific server) and confirm that the contents of the session were not altered.

Trusted Execution Environment (TEE) attestations: A trusted execution environment (TEE) is a sandboxed computational environment that is isolated from the operational processes of its host system. TEEs ensure that whatever application code or data stored/used in the computation environment retains integrity, confidentiality, and immutability. Users can also generate an attestation to prove an application instance is running within the trusted execution environment.

Certain classes of decentralized oracles require oracle node operators to provide TEE attestations. This confirms to a user that the node operator is running an instance of oracle client in a trusted execution environment. TEEs prevent external processes from altering or reading an application's code and data, hence, those attestations prove that the oracle node has kept the information intact and confidential.

Consensus-based validation of information {#consensus-based-validation-of-information}

Centralized oracles rely on a single source of truth when providing data to smart contracts, which introduces the possibility of publishing inaccurate information. Decentralized oracles solve this problem by relying on multiple oracle nodes to query off-chain information. By comparing data from multiple sources, decentralized oracles reduce the risk of passing invalid information to on-chain contracts.

Decentralized oracles, however, must deal with discrepancies in information retrieved from multiple off-chain sources. To minimize differences in information and ensure the data passed to the oracle contract reflects the collective opinion of oracle nodes, decentralized oracles use the following mechanisms:

Voting/staking on accuracy of data

Some decentralized oracle networks require participants to vote or stake on the accuracy of answers to data queries (e.g., "Who won the 2020 US election?") using the network's native token. An aggregation protocol then aggregates the votes and stakes and takes the answer supported by the majority as the valid one.

Nodes whose answers deviate from the majority answer are penalized by having their tokens distributed to others who provide more correct values. Forcing nodes to provide a bond before providing data incentivizes honest responses since they are assumed to be rational economic actors intent on maximizing returns.

Staking/voting also protects decentralized oracles from "Sybil attacks" where malicious actors create multiple identities to game the consensus system. However, staking cannot prevent "freeloading" (oracle nodes copying information from others) and "lazy validation" (oracle nodes following the majority without verifying the information themselves).

Schelling point mechanisms

Schelling point is a game-theory concept that assumes multiple entities will always default to a common solution to a problem in absence of any communication. Schelling-point mechanisms are often used in decentralized oracle networks to enable nodes reach consensus on answers to data requests.

An early idea for this was SchellingCoin, a proposed data feed where participants submit responses to "scalar" questions (questions whose answers are described by magnitude, e.g., "what is the price of ETH?"), along with a deposit. Users who

provide values between the 25th and 75th percentile are rewarded, while those whose values deviate largely from the median value are penalized.

While SchellingCoin doesn't exist today, a number of decentralized oracles—notably Maker Protocol's Oracles—use the schelling-point mechanism to improve accuracy of oracle data. Each Maker Oracle consists of an off-chain P2P network of nodes ("relayers" and "feeds") who submit market prices for collateral assets and an on-chain "Medianizer" contract that calculates the median of all provided values. Once the specified delay period is over, this median value becomes the new reference price for the associated asset.

Other examples of oracles that use Schelling point mechanisms include Chainlink Off-Chain Reporting and Witnet. In both systems, responses from oracle nodes in the peer-to-peer network are aggregated into a single aggregate value, such as a mean or median. Nodes are rewarded or punished according to the extent to which their responses align with or deviate from the aggregate value.

Schelling point mechanisms are attractive because they minimize on-chain footprint (only one transaction needs to be sent) while guaranteeing decentralization. The latter is possible because nodes must sign off on the list of submitted responses before it is fed into the algorithm that produces the mean/median value.

Availability {#availability}

Decentralized oracle services ensure high availability of off-chain data to smart contracts. This is achieved by decentralizing both the source of off-chain information and nodes responsible for transferring the information on-chain.

This ensures fault-tolerance since the oracle contract can rely on multiple nodes (who also rely on multiple data sources) to execute queries from other contracts. Decentralization at the source *and* node-operator level is crucial—a network of oracle nodes serving information retrieved from the same source will run into the same problem as a centralized oracle.

It is also possible for stake-based oracles can slash node operators who fail to respond quickly to data requests. This significantly incentivizes oracle nodes to invest in fault-tolerant infrastructure and provide data in timely fashion.

Good incentive compatibility {#good-incentive-compatibility}

Decentralized oracles implement various incentive designs to prevent Byzantine behavior among oracle nodes. Specifically, they achieve *attributability* and *accountability*:

1. Decentralized oracle nodes are often required to sign the data they provide in response to data requests. This information helps with evaluating the historical performance of oracle nodes, such that users can filter out unreliable oracle nodes when making data requests. An example is Witnet's Algorithmic Reputation System.
2. Decentralized oracles—as explained earlier—may require nodes to place a stake on their confidence in the truth of data they submit. If the claim checks out, this stake can be returned along with rewards for honest service. But it can also be slashed in case the information is incorrect, which provides some measure of accountability.

Applications of oracles in smart contracts {#applications-of-oracles-in-smart-contracts}

The following are common use-cases for oracles in Ethereum:

Retrieving financial data {#retrieving-financial-data}

Decentralized finance (DeFi) applications allow for peer-to-peer lending, borrowing, and trading of assets. This often requires getting different financial information, including exchange rate data (for calculating the fiat value of cryptocurrencies or comparing token prices) and capital markets data (for calculating the value of tokenized assets, such as gold or the US dollar).

A DeFi lending protocol, for example, needs to query current market prices for assets (e.g., ETH) deposited as collateral. This lets the contract determine the value of collateral assets and determine how much it can borrow from the system.

Popular “price oracles” (as they are often called) in DeFi include Chainlink Price Feeds, Compound Protocol’s Open Price Feed, Uniswap’s Time-Weighted Average Prices (TWAPs), and Maker Oracles.

Builders should understand the caveats that come with these price oracles before integrating them into their project. This article provides a detailed analysis of what to consider when planning to use any of the price oracles mentioned.

Below is an example of how you can retrieve the latest ETH price in your smart contract using a Chainlink price feed:

```solidity pragma solidity ^0.6.7;

import "@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol";

contract PriceConsumerV3 {

```
AggregatorV3Interface internal priceFeed;

/**
 * Network: Kovan
 * Aggregator: ETH/USD
 * Address: 0x9326BFA02ADD2366b30bacB125260Af641031331
 */
constructor() public {
    priceFeed = AggregatorV3Interface(0x9326BFA02ADD2366b30bacB125260Af641031331);
}

/**
 * Returns the latest price
 */
function getLatestPrice() public view returns (int) {
    (
        uint80 roundID,
        int price,
        uint startedAt,
        uint timeStamp,
        uint80 answeredInRound
) = priceFeed.latestRoundData();
    return price;
}
```

} ```

## Generating verifiable randomness {#generating-verifiable-randomness}

Certain blockchain applications, such as blockchain-based games or lottery schemes, require a high level of unpredictability and randomness to work effectively. However, the deterministic execution of blockchains eliminates randomness.

The usual approach is to use pseudorandom cryptographic functions, such as `blockhash`, but these can be manipulated by miners solving the proof-of-work algorithm. Also, Ethereum’s switch to proof-of-stake means developers can no longer rely on `blockhash` for on-chain randomness (the Beacon Chain’s RANDAO mechanism provides an alternative source of randomness, though).

It is possible to generate the random value off-chain and send it on-chain, but doing so imposes high trust requirements on users. They must believe the value was truly generated via unpredictable mechanisms and wasn’t altered in transit.

Oracles designed for off-chain computation solve this problem by securely generating random outcomes off-chain that they broadcast on-chain along with cryptographic proofs attesting to the unpredictability of the process. An example is Chainlink VRF (Verifiable Random Function), which is a provably fair and tamper-proof random number generator (RNG) useful for building reliable smart contracts for applications that rely on unpredictable outcomes. Another example is API3 QRNG that serves Quantum random number generation (QRNG) is a public method of Web3 RNG based on quantum phenomena, served with the courtesy of the Australian National University (ANU).

## Getting outcomes for events {#getting-outcomes-for-events}

With oracles, creating smart contracts that respond to real-world events is easy. Oracle services make this possible by allowing contracts to connect to external APIs through off-chain components and consume information from those data sources. For example, the prediction dapp mentioned earlier may request an oracle to return election results from a trusted
```

off-chain source (e.g., the Associated Press).

Using oracles to retrieve data based on real-world outcomes enables other novel use cases; for example, a decentralized insurance product needs accurate information about weather, disasters, etc. to work effectively.

Automating smart contracts {#automating-smart-contracts}

Smart contracts do not run automatically; rather, an externally owned account (EOA), or another contract account, must trigger the right functions to execute the contract's code. In most cases, the bulk of the contract's functions are public and can be invoked by EOAs and other contracts.

But there are also *private functions* within a contract that are inaccessible to others;, but that are critical to a dapp's overall functionality. Examples include a `mintERC721Token()` function that periodically mints new NFTs for users, a function for awarding payouts in a prediction market, or a function for unlocking staked tokens in a DEX.

Developers will need to trigger such functions at intervals to keep the application running smoothly. However, this might lead to more hours lost on mundane tasks for developers, which is why automating execution of smart contracts is attractive.

Some decentralized oracle networks offer automation services, which allow off-chain oracle nodes to trigger smart contract functions according to parameters defined by the user. Typically, this requires "registering" the target contract with the oracle service, providing funds to pay the oracle operator, and specifying the conditions or times to trigger the contract.

Chainlink's Keeper Network provides options for smart contracts to outsource regular maintenance tasks in a trust minimized and decentralized manner. Read the official Keeper's documentation for information on making your contract Keeper-compatible and using the Upkeep service.

How to use blockchain oracles {#use-blockchain-oracles}

There are multiple oracle applications you can integrate into your Ethereum dapp:

Chainlink - *Chainlink decentralized oracle networks provide tamper-proof inputs, outputs, and computations to support advanced smart contracts on any blockchain.*

Witnet - *Witnet is a permissionless, decentralized, and censorship-resistant oracle helping smart contracts to react to real world events with strong crypto-economic guarantees.*

UMA Oracle - *UMA's optimistic oracle allows smart contracts to quickly and receive any kind of data for different applications, including insurance, financial derivatives, and prediction markets.*

Tellor - *Tellor is a transparent and permissionless oracle protocol for your smart contract to easily get any data whenever it needs it.*

Band Protocol - *Band Protocol is a cross-chain data oracle platform that aggregates and connects real-world data and APIs to smart contracts.*

Paralink - *Paralink provides an open source and decentralized oracle platform for smart contracts running on Ethereum and other popular blockchains.*

Pyth Network - *The Pyth network is a first-party financial oracle network designed to publish continuous real-world data on-chain in a tamper-resistant, decentralized, and self-sustainable environment.*

API3 DAO - *API3 DAO is delivering first-party oracle solutions that deliver greater source transparency, security and scalability in a decentralized solution for smart contracts*

Further reading {#further-reading}

Articles

- What Is a Blockchain Oracle? — *Chainlink*

- What is a Blockchain Oracle? — *Patrick Collins*
- Decentralised Oracles: a comprehensive overview — *Julien Thevenard*
- Implementing a Blockchain Oracle on Ethereum – *Pedro Costa*
- Why can't smart contracts make API calls? — *StackExchange*
- Why we need decentralized oracles — *Bankless*
- So you want to use a price oracle — *samczsun*

Videos

- Oracles and the Expansion of Blockchain Utility — *Real Vision Finance*
- The differences between first party and third party oracles - *Blockchain Oracle Summit*

Tutorials

- How to Fetch the Current Price of Ethereum in Solidity — *Chainlink*

Example projects

- Full Chainlink starter project for Ethereum in Solidity — *HackBG*