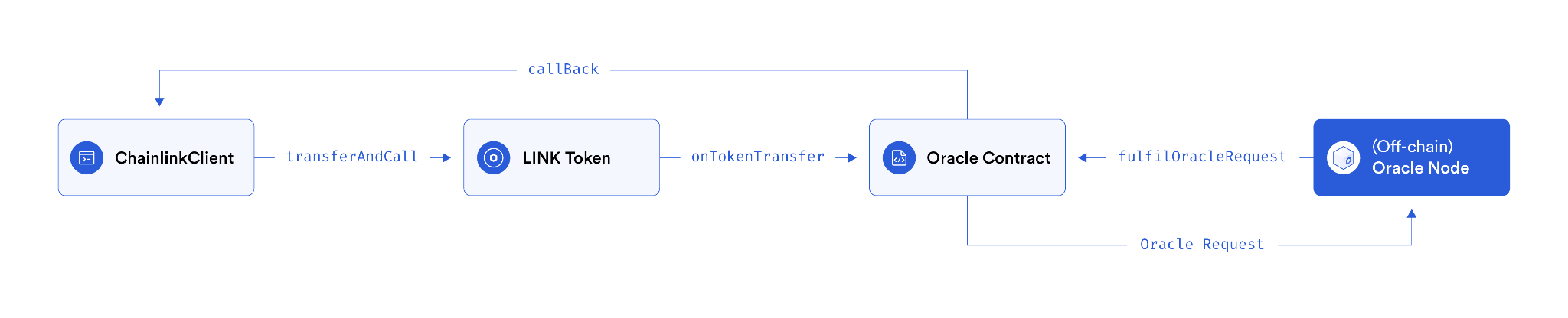
**Architecture Overview**

# **Basic Request Model**

**Chainlink connects smart contracts** with external data using its **decentralized oracle network.** Chainlink API requests are **handled 1:1 by an oracle.**

The [Basic Request Model](https://docs.chain.link/docs/architecture-request-model/) describes the **on-chain architecture of requesting data from a single oracle source.**

****

## **ChainlinkClient**

[ChainlinkClient](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.6/ChainlinkClient.sol) is a **parent contract that enables smart contracts** to consume data from **oracles**. It's available in the **Chainlink smart contract library** which can be [installed using the latest package managers](https://docs.chain.link/docs/create-a-chainlinked-project/).

The **client constructs and makes a request** to a known Chainlink oracle through the **transferAndCall function,** implemented by the **LINK** token. This request contains encoded information that is required for the cycle to succeed.

In the ChainlinkClient contract, this call is initiated with a call to **sendChainlinkRequestTo**.

## **LINK Token**

**LINK is an** [**ERC-677**](https://github.com/ethereum/EIPs/issues/677) **compliant token** which implements **transferAndCall**, a function that allows tokens to be transferred whilst also **triggering logic in the receiving contract within** a **single transaction.**

## 

## **Oracle Contract**

[**Oracle**](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.6/Oracle.sol) **contracts** are owned by **oracle node operators**, which run alongside off-chain oracle nodes.

### **Request**

The client contract that initiates this cycle must create a request with:

* the **oracle address**
* the **job ID**, so the oracle know what tasks to perform
* the **callback function**, which the oracle will send the response to.

**Oracle contracts** are responsible for handling **on-chain requests made through the LINK token**, by implementing **onTokenTransfer as a** [**LinkTokenReceiver**](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.6/LinkTokenReceiver.sol).

Upon execution of this function, the **oracle contract emits an OracleRequest** event containing information about the request. This event is crucial, as it is **monitored by the off-chain oracle node which acts upon it.**

### **Fulfillment**

For fulfillment, the oracle contract has a **fulfillOracleRequest function** which is used by the node to fulfill a request once it has the result of the job. This function returns the result to the **ChainlinkClient** using the callback function defined in the original request.

## **Off-Chain** **Oracle** **Node**

The off-chain oracle node is responsible for **listening for events emitted by its corresponding on-chain smart contract.** Once it detects an OracleRequest event, it uses the data emitted to perform a job.

The most common job type for a **Node is to make a GET request to an API, retrieve some data from it, parse the response, convert the result into blockchain compatible data, then submit it in a transaction back to the oracle contract, using the fulfillOracleRequest function.**

# **Decentralized Data Model**

For a more robust and trustworthy answer, you can **aggregate data from many oracles.**

With **on-chain aggregation, data is aggregated from a decentralized network of independent oracle nodes.**

This architecture is applied to **Chainlink Data Feeds**, which can aggregate data such as asset price data.

The [Decentralized Data Model](https://docs.chain.link/docs/architecture-decentralized-model/) describes **how data is aggregated, and how consumer contracts can retrieve this data.**

**Data Aggregation**

Each data feed is updated by multiple, independent Chainlink oracle operators. Aggregation is handled on-chain by [**FluxAggregator**](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.6/FluxAggregator.sol).

**Shared Data Resource**

Each **data feed** is built and funded by the community of users who rely on accurate, up-to-date data in their smart contracts. As more users rely on and contribute to a data feed, the quality of the data feed improves. For this reason, **each data feed has its own properties depending on the needs of its community of users.**

## **Decentralized Oracle Network**

Each data feed is updated by a **decentralized oracle network**.

Each **oracle operator is rewarded for publishing data.**

The number of oracles contributing to each feed varies. In order for an update to take place, the data feed contract must receive responses from a **minimum number of oracles or the latest answer will not be updated.**

Each oracle in the set **publishes data during an aggregation round.** That data is validated and aggregated by a smart contract, which forms the feed's latest and trusted answer.

Developers wishing to use an asset's latest and trusted answer can do so easily by following the example from the [Get the Latest Price](https://docs.chain.link/docs/get-the-latest-price/) page.

## 

## **Aggregation Parameters**

The **Deviation Threshold and Heartbeat Threshold** are parameters that can **trigger price feeds to update during an aggregation round.** Each aggregation round triggers based on one of these parameters. The first condition that is met triggers an update to the data.

### **Deviation Threshold**

A **new aggregation round starts when a node identifies that the off-chain values** deviate by more than the **defined deviation threshold from the on-chain value.** Individual nodes monitor one or more data providers for each feed.

### **Heartbeat Threshold**

A **new aggregation round starts after a specified amount of time from the last update.**

**Consumer**

Consumer contracts simply reference the correct [**AggregatorV3Interface**](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol) and call one of the exposed functions.

**Proxy**

Proxy contracts are **on-chain proxies that store the most up-to-date Aggregator** for a particular data feed. Using proxies enables the underlying Aggregator to be upgraded without any interruption of service for consuming contracts.

**Aggregators**

Aggregators are the contracts that receive periodic data updates from multiple **Oracles**. They aggregate and store **data on-chain so that consumers can retrieve it** and act upon it within the same transaction.

This data can be accessed by referencing the Data Feed address using the [**AggregatorV3Interface**](https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol) contract.

**Off-Chain Reporting (OCR)**

Off-Chain Reporting (OCR) is the next **Oracle network upgrade** replacing the [**FluxAggregator model**](https://docs.chain.link/docs/architecture-decentralized-model/)**.**

It represents a significant step towards increasing the **decentralization and scalability of Chainlink networks.**

In the [**FluxAggregator model**](https://docs.chain.link/docs/architecture-decentralized-model/),

* every node must submit their **price value individually,** and once **all responses are received on-chain, the contract aggregates them to confirm the price.**

This solution has been highly effective and reliable since its release. However, there are some drawbacks.

Since **every single node must submit a transaction per round, each pays gas to do so**, with aggregation occurring on-chain once all nodes have submitted.

With our Off-Chain Reporting aggregators,

* all nodes communicate using a **peer to peer network.**
* During the communication process, a **lightweight consensus algorithm** is run, in which **every single node reports its price observation and signs it.**
* A **single, aggregate transaction is then transmitted, saving a significant amount of gas.**

The OCR protocol allows nodes to **aggregate their observations into a single report off-chain using a secure P2P network.**

A single node then **submits a transaction with the aggregated report to the chain.**

Each report consists of **many nodes' observations** and has to be **signed by a quorum of nodes.** These **signatures are verified on-chain.**

Since only one transaction is submitted per round, the following benefits are achieved:

* **Overall network congestion from Chainlink oracle networks** is reduced dramatically
* **Individual node operators spend far less on gas costs**
* **Node networks are more scalable** because data feeds can accommodate more nodes
* **Data feeds can be updated in a more timely manner** since each round needn't wait for multiple transactions to be confirmed before a price is confirmed on-chain.

**How does it work?**

Protocol execution mostly happens **off-chain over a peer to peer network** between Chainlink nodes.

The nodes regularly **elect a new leader node who drives the rest of the protocol.**

* The leader regularly requests **followers to provide freshly signed observations and aggregates them into a report.**
* It then sends this **report back to the followers and asks them to verify the report's validity.**
* If a **quorum of followers approves the report** by sending a signed copy back to the leader, the leader assembles a final report with the quorum's signatures and broadcasts it to all followers.
* The nodes then **attempt to transmit the final report to the smart contract** according to a randomized schedule.
* Finally, the smart contract verifies that a **quorum of nodes** signed the report and exposes the median value to consumers.

# **Connect Your Smart Contracts to the Outside World**

**Chainlink Data Feeds** are the quickest way to connect your **smart contracts to the real-world market prices of assets**.

For example, one use for data feeds is to **enable smart contracts to retrieve the latest pricing data of an asset in a single call.**

**In Solidity -** To consume price data, your smart contract should reference [AggregatorV3Interface](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol), which defines the external functions implemented by Data Feeds.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

**import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";**

contract PriceConsumerV3 {

**AggregatorV3Interface internal priceFeed;**

/\*\*

\* Network: Kovan

\* Aggregator: ETH/USD

\* Address: 0x9326BFA02ADD2366b30bacB125260Af641031331

\*/

constructor() {

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;

}

}

Data Feeds are updated in **rounds**.

Rounds are identified by their **roundId**, which increases with each new round.

The increase might not be monotonic.

Knowing the **roundId of a previous round allows contracts to consume historical price data.**

The most common use case for Data Feeds is to [**Get the Latest Price**](https://docs.chain.link/docs/get-the-latest-price/)**.**

There are **two parameters** that can cause Chainlink nodes to update:

| **Name** | **Description** |
| --- | --- |
| **Deviation Threshold** | Chainlink nodes are **monitoring prices of assets off-chain.** The deviation of the real-world price of an asset beyond a certain interval triggers all the nodes to update. |
| **Heartbeat Threshold** | If the price stays within the deviation parameters, it will **only trigger an update every *X* minutes / hours.** |

The updatedAt data feed property is the timestamp of an answered round and answeredInRound is the round it was updated in.

If answeredInRound is less than roundId, the answer is being carried over. If answeredInRound is equal to roundId, then the answer is fresh.

The Chainlink Feed Registry is an on-chain mapping of assets to feeds. It enables you to query Chainlink data feeds from asset addresses directly, without needing to know the feed contract addresses. They enable smart contracts to get the latest price of an asset in a single call, from a single contract.

To consume price data from the Feed Registry, your smart contract should reference [FeedRegistryInterface](https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/interfaces/FeedRegistryInterface.sol), which defines the external functions implemented by the Feed Registry.

## **Random Number Consumer**

Chainlink VRF follows the [**Request & Receive Data**](https://docs.chain.link/docs/request-and-receive-data/) **cycle**. To consume randomness, your contract should inherit from [**VRFConsumerBase**](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.8/VRFConsumerBase.sol) and define **two required functions:**

* **requestRandomness**, which makes the **initial request** for randomness.
* **fulfillRandomness**, which is the **function that receives and does something** with verified randomness.