

```
# api-reference.mdx:
```

```
---
```

```
section: nodeOperator
date: Last Modified
title: "ChainlinkClient API Reference"
isMdx: true
---
```

```
import { Aside, CodeSample } from "@components"
import AnyApiCallout from "@features/any-api/common/AnyApiCallout.astro"
```

```
<Aside type="note" title="API reference for ChainlinkClient">
```

```
ChainlinkClient contracts
can communicate with Operator contracts.
```

```
</Aside>
```

Index

Methods

Name	Description
:	:
:	:
:	:
:	:
:	:
setChainlinkOracle	Sets the stored address for the oracle contract
setChainlinkToken	Sets the stored address for the LINK token
buildChainlinkRequest	Instantiates a Request object with the required parameters
buildOperatorRequest	Instantiates a Request object with the required parameters. Note the oracle must be an Operator contract
sendChainlinkRequest	Sends the request payload to the stored address stored as chainlinkOracleAddress
sendChainlinkRequestTo	Sends a request to the oracle address specified
sendOperatorRequest	Sends the request payload to the stored address stored as chainlinkOracleAddress. Note the oracle must be an Operator contract
sendOperatorRequestTo	Sends a request to the oracle address specified. Note the oracle must be an Operator contract
validateChainlinkCallback	Secures the fulfillment callback to make sure it is only called by permissioned senders
addChainlinkExternalRequest	Allows a Chainlinked contract to track unfulfilled requests that it hasn't created itself
cancelChainlinkRequest	Cancels Chainlink requests attempting to contact an unresponsive node

useChainlinkWithENS	Looks up the addresses of the LINK token and Oracle contract through ENS
updateChainlinkOracleWithENS	Updates the stored oracle address with the latest address resolved through ENS
chainlinkTokenAddress	Returns the stored address of the LINK token
chainlinkOracleAddress	Returns the stored address of the oracle contract

Events

Name	Description
:-----	
ChainlinkRequested	Emitted from a Chainlinked contract when a request is sent to an oracle
ChainlinkFulfilled	Emitted from a Chainlinked contract when a request is fulfilled by an oracle
ChainlinkCancelled	Emitted from a Chainlinked contract when a request is cancelled

Modifiers

Name	Description
:-----	
recordChainlinkFulfillment	Used on fulfillment callbacks to ensure that the caller and requestId are valid. This is the modifier equivalent of the method validateChainlinkCallback

Constants

Name	Description
:-----	
- LINKDIVISIBILITY	Helper uint256 to represent the divisibility of a LINK token. Equivalent to 10 ¹⁸

Structs

Name	Description
:-----	
Chainlink.Request	All of the parameters that can be passed over in a Chainlink request

Methods

Below you'll find each helper explained in greater detail alongside respective implementation examples to help you leverage these methods once you start building your own Chainlinked contract.

After the function signature and a short description, two code examples are

provided, one focusing on the exact usage of the method and one where the helper is presented in the context of a full contract.

setChainlinkOracle

```
{/ prettier-ignore /}  
solidity  
function setChainlinkOracle(  
    address oracle  
)
```

Sets a private storage variable provided for convenience if your contract only needs to talk to one oracle and you do not want to specify it on every request. Once an oracle is set with setChainlinkOracle that is the address used with sendChainlinkRequest.

Retrieve the oracle address using chainlinkOracleAddress. These getters and setters are provided to enforce that changes to the oracle are explicitly made in the code.

```
{/ prettier-ignore /}  
solidity  
constructor(address oracle)  
{  
    setChainlinkOracle(oracle);  
}
```

setChainlinkToken

```
{/ prettier-ignore /}  
solidity  
setChainlinkToken(  
    address link  
)
```

Sets the stored address for the LINK token which is used to send requests to Oracles. There are different token addresses on different network. See LINK Token Contracts for the address of the LINK token on the network you're deploying to.

```
{/ prettier-ignore /}  
solidity  
constructor(address link)  
    public  
{  
    setChainlinkToken(link);  
}
```

buildChainlinkRequest

Use buildOperatorRequest function if the oracle is an Operator contract.

```
{/ prettier-ignore /}  
solidity  
function buildChainlinkRequest(  
    bytes32 jobId,  
    address callbackAddress,  
    bytes4 callbackFunctionSignature  
) returns (Chainlink.Request memory request)
```

Instantiates a Request from the Chainlink contract. A Request is a struct which contains the necessary parameters to be sent to the oracle contract. The buildChainlinkRequest function takes an ID, which can be a Job ID, a callback address to receive the resulting data, and a callback function signature to call on the callback address.

```

{
  // prettier-ignore
}
solidity
function requestPrice()
  public
{
  bytes32 jobId = "493610cff14346f786f88ed791ab7704";
  bytes4 selector = this.myCallback.selector;
  // build a request that calls the myCallback function defined
  // below by specifying the address of this contract and the function
  // selector of the myCallback
  Chainlink.Request memory request = buildChainlinkRequest(
    jobId,
    address(this),
    selector);
}

```

buildOperatorRequest

This function is similar to buildChainlinkRequestfunction. One major difference is that buildOperatorRequest does not allow setting up the address of the callback. The callback address is set to the address of the calling contract.

It is recommended to use buildOperatorRequest but make sure the oracle you are contacting is an Operator contract.

```

{
  // prettier-ignore
}
solidity
function buildOperatorRequest(
  bytes32 jobId,
  bytes4 callbackFunctionSignature
) returns (Chainlink.Request memory request)

```

Instantiates a Request from the Chainlink contract. A Request is a struct that contains the necessary parameters to be sent to the oracle contract. The buildOperatorRequest function takes an ID, which can be a Job ID, and a callback function signature to call on the calling contract address.

```

{
  // prettier-ignore
}
solidity
function requestPrice()
  public
{
  bytes32 jobId = "493610cff14346f786f88ed791ab7704";
  bytes4 selector = this.myCallback.selector;
  // build a request that calls the myCallback function defined
  // below by specifying the function selector of myCallback
  Chainlink.Request memory request = buildOperatorRequest(
    jobId,
    selector);
}

```

sendChainlinkRequest

Use `sendOperatorRequest` function if the oracle is an Operator contract.

```
{/ prettier-ignore /}
solidity
function sendChainlinkRequest(
    Chainlink.Request memory req,
    uint256 payment
) returns (bytes32 requestId)
```

Sends the request payload to the stored oracle address. It takes a `Chainlink.Request` and the amount of LINK to send amount as parameters. The request is serialized and calls `oracleRequest` on the address stored in `chainlinkOracleAddress` via the LINK token's `transferAndCall` method.

`sendChainlinkRequest` returns the ID of the request. If your application needs to, your contract can store that ID, but you don't need to. The `ChainlinkClient` helpers will store the ID under the hood, along with the oracle address, and use them when you call `recordChainlinkFulfillment` in your callback function to make sure only that the address you want can call your Chainlink callback function.

`sendChainlinkRequest` emits a `ChainlinkRequested` event containing the request ID, if you would like to use it in your Web3 application.

```
{/ prettier-ignore /}
solidity
function requestPrice()
    public
{
    Chainlink.Request memory request = buildChainlinkRequest(jobId, address(this),
this.callback.selector);
    uint256 paymentAmount = 1 LINKDIVISIBILITY / 10; // Equivalent to 0.1 LINK

    // send the request that you just built
    sendChainlinkRequest(request, paymentAmount);
}
```

sendChainlinkRequestTo

Use `sendOperatorRequestTo` function if the oracle is an Operator contract.

```
{/ prettier-ignore /}
solidity
function sendChainlinkRequestTo(
    address oracle,
    Chainlink.Request memory req,
    uint256 payment
) returns (bytes32 requestId)
```

Similar to `sendChainlinkRequest`, `sendChainlinkRequestTo` sends a Request but allows the target oracle to be specified. It requires an address, a Request, and an amount, and returns the requestId. This allows a requesting contract to create and track requests sent to multiple oracle contract addresses.

`sendChainlinkRequestTo` emits a `ChainlinkRequested` event containing the request ID, if you would like to use it in your Web3 application.

```
{/ prettier-ignore /}
```

```

solidity
function requestPriceFrom(address oracle)
    public
{
    Chainlink.Request memory request = buildChainlinkRequest(jobId, address(this),
this.callback.callbackSelector);
    uint256 paymentAmount = 1 LINKDIVISIBILITY; // = 1 LINK

    // send the request that you just built to a specified oracle
    sendChainlinkRequestTo(oracle, request, paymentAmount);
}

```

sendOperatorRequest

This function is similar to sendChainlinkRequestfunction. It is recommended to use sendOperatorRequest but make sure the oracle you are contacting is an Operator contract.

```

{/ prettier-ignore /}
solidity
function sendOperatorRequest(
    Chainlink.Request memory req,
    uint256 payment
) returns (bytes32 requestId)

```

The sendOperatorRequest function sends the request payload to the stored oracle address. It takes a Chainlink.Request and the amount of LINK to send amount as parameters. The request is serialized and calls operatorRequest on the address stored in chainlinkOracleAddress using the LINK token's transferAndCall method.

sendOperatorRequest returns the ID of the request. Optionally, your contract can store the ID if your application needs it. The ChainlinkClient helpers store the ID and the oracle address and use them when you call recordChainlinkFulfillment in your callback function. This ensures that only the specified address can call your Chainlink callback function.

sendOperatorRequest emits a ChainlinkRequested event containing the request ID that you can use in your Web3 application.

```

{/ prettier-ignore /}
solidity
function requestPrice()
    public
{
    Chainlink.Request memory request = buildOperatorRequest(jobId,
this.callback.selector);
    uint256 paymentAmount = 1 LINKDIVISIBILITY / 10; // Equivalent to 0.1 LINK

    // send the request that you just built
    sendOperatorRequest(request, paymentAmount);
}

```

sendOperatorRequestTo

This function is similar to sendChainlinkRequestTofunction. It is recommended to use sendOperatorRequestTo, but make sure the oracle you are contacting is an Operator contract.

```

{/ prettier-ignore /}

```

```

solidity
function sendChainlinkRequestTo(
    address oracle,
    Chainlink.Request memory req,
    uint256 payment
) returns (bytes32 requestId)

```

Similar to `sendOperatorRequest`, `sendOperatorRequestTo` sends a `Request` but allows the target oracle to be specified. It requires an address, a `Request`, and an amount, and returns the `requestId`. This allows a requesting contract to create and track requests sent to multiple oracle contract addresses.

`sendOperatorRequestTo` emits a `ChainlinkRequested` event containing a request ID that you can use in your Web3 application.

```

{/ prettier-ignore /}
solidity
function requestPriceFrom(address oracle)
    public
{
    Chainlink.Request memory request = buildOperatorRequest(jobId,
this.callback.callbackSelector);
    uint256 paymentAmount = 1 LINKDIVISIBILITY; // = 1 LINK

    // send the request that you just built to a specified oracle
    sendOperatorRequestTo(oracle, request, paymentAmount);
}

```

`validateChainlinkCallback`

```

{/ prettier-ignore /}
solidity
function validateChainlinkCallback(
    bytes32 requestId
)

```

Used on fulfillment callbacks to ensure that the caller and `requestId` are valid. They protect `ChainlinkClient` callbacks from being called by malicious callers. `validateChainlinkCallback` allows for a request to be called

This is the method equivalent of the modifier `recordChainlinkFulfillment`. Either `validateChainlinkCallback` or `recordChainlinkFulfillment` should be used on all fulfillment functions to ensure that the caller and `requestId` are valid. Use the modifier or the method, not both.

`validateChainlinkCallback` emits a `ChainlinkFulfilled` event.

```

{/ prettier-ignore /}
solidity
function myCallback(bytes32 requestId, uint256 price)
    public
{
    validateChainlinkCallback(requestId);
    currentPrice = price;
}

```

<Aside type="caution" title="Do not call multiple times">

Do not call `validateChainlinkCallback` multiple times. The nature of validating the callback is to ensure the response is only received once and not replayed.

Calling a second time with the same method ID will trigger a revert. Similarly, your callback should validate using either `validateChainlinkCallback` or `recordChainlinkFulfillment`, not both.

</Aside>

`addChainlinkExternalRequest`

```
{/ prettier-ignore /}  
solidity  
function addChainlinkExternalRequest(  
    address oracle,  
    bytes32 requestId  
)
```

`addChainlinkExternalRequest` allows a Chainlink contract to track unfulfilled requests that it hasn't created itself. For example, contract A creates a request and sets the callback for contract B. Contract B needs to know about the request created by contract A so that it can validate the callback when it is executed.

```
{/ prettier-ignore /}  
solidity  
function expectResponseFor(bytes32 requestId)  
    public  
{  
    addChainlinkExternalRequest(chainlinkOracleAddress(), requestId);  
}
```

<Aside type="caution" title="Be careful adding external requests">

Being able to change a request means that you can change the data fed into a contract. Permissioning someone to make external requests can allow them to change the outcome of your contract. You should be sure to make sure that they are a trusted to do so. If they are not trusted to do so, you should put the request making logic onchain where it is auditable and tamperproof.

</Aside>

`cancelChainlinkRequest`

```
{/ prettier-ignore /}  
solidity  
function cancelChainlinkRequest(bytes32 requestId,  
    uint256 payment,  
    bytes4 callbackFunc,  
    uint256 expiration  
)
```

In case an oracle node does not respond, it may be necessary to retrieve the LINK used to pay for the unfulfilled request. The `cancelChainlinkRequest` will send the cancel request to the address used for the request, which transfers the amount of LINK back to the requesting contract, and delete it from the tracked requests.

The default expiration for a request is five minutes, after which it can be cancelled. The cancellation must be sent by the address which was specified as the callback location of the contract.

For the sake of efficient gas usage, only a hash of the request's parameters are stored onchain. In order to validate the terms of the request and that it can be

calculated, the request parameters must be provided. Additionally, cancellation must be called by the address which the callback would otherwise have been called on.

cancelChainlinkRequest emits a ChainlinkCancelled event.

```

{
  // prettier-ignore
}
solidity
function cancelRequest(
    bytes32 requestId,
    uint256 payment,
    bytes4 callbackFunc,
    uint256 expiration
) public {
    cancelChainlinkRequest(requestId, payment, callbackFunc, expiration);
}

```

useChainlinkWithENS

```

{
  // prettier-ignore
}
solidity
function useChainlinkWithENS(
    address ens,
    bytes32 node
)

```

Allows a Chainlink contract to store the addresses of the LINK token and oracle contract addresses without supplying the addresses themselves. We use ENS where available to resolve these addresses. It requires the address of the ENS contract and the node (which is a hash) for the domain.

If your Oracle provider supports using ENS for rolling upgrades to their oracle contract, once you've pointed your Chainlinked contract to the ENS records then you can update the records using updateChainlinkOracleWithENS.

```

{
  // prettier-ignore
}
solidity
address constant ROPSTENENS = 0x112234455C3a32FD11230C42E7Bccd4A84e02010;
bytes32 constant ROPSTENCHAINLINKENS =
0xead9c0180f6d685e43522fcfe277c2f0465fe930fb32b5b415826eacf9803727;

constructor(){
    useChainlinkWithENS(ROPSTENENS, ROPSTENCHAINLINKENS);
}

```

<Aside type="caution" title="Updating oracle addresses">

If an oracle provider supports listing their oracle on ENS, that provides the added security of being able to update any issues that may arise. The tradeoff here is that by using their ENS record, you are allowing whoever controls that record and the corresponding code it points to. If your contract does this, you must either audit the updated code and make sure it matches Operator.sol or trust whoever can update the records.

</Aside>

updateChainlinkOracleWithENS

```

{
  // prettier-ignore
}
solidity
function updateChainlinkOracleWithENS()

```

Updates the stored oracle contract address with the latest address resolved through the ENS contract. This requires the oracle provider to support listing their address on ENS.

This method only works after useChainlinkWithENS has been called on the contract.

```

{
  // prettier-ignore
  solidity
  function updateOracleAddressToLatest() public {
    updateChainlinkOracleWithENS();
  }
}

```

<Aside type="caution" title="Updating oracle addresses">

If an oracle provider supports listing their oracle on ENS, that provides the added security of being able to update any issues that may arise. The tradeoff here is that by using their ENS record, you are allowing whoever controls that record and the corresponding code it points to. If your contract does this, you must either audit the updated code and make sure it matches Operator.sol or trust whoever can update the records.

</Aside>

chainlinkTokenAddress

```

{
  // prettier-ignore
  solidity
  function chainlinkTokenAddress() returns (address)
}

```

The chainlinkTokenAddress function is a helper used to return the stored address of the Chainlink token. This variable is protected and so only made available through getters and setters.

```

{
  // prettier-ignore
  solidity
  function withdrawLink() public {
    LinkTokenInterface link = LinkTokenInterface(chainlinkTokenAddress());

    require(link.transfer(msg.sender, link.balanceOf(address(this))), "Unable to transfer");
  }
}

```

chainlinkOracleAddress

The chainlinkOracleAddress function is a helper used to return the stored address of the oracle contract.

```

{
  // prettier-ignore
  solidity
  function getOracle() public view returns (address) {
    return chainlinkOracleAddress();
  }
}

```

Events

ChainlinkRequested

```

{/ prettier-ignore /}
solidity
event ChainlinkRequested(
    bytes32 indexed id
)

```

Emitted when `sendChainlinkRequest` and `sendChainlinkRequestTo` are called. Includes the request ID as an event topic.

ChainlinkFulfilled

```

{/ prettier-ignore /}
solidity
event ChainlinkFulfilled(
    bytes32 indexed id
)

```

Emitted when `validateChainlinkCallback` or `recordChainlinkFulfillment` are called. Includes the request ID as an event topic.

ChainlinkCancelled

```

{/ prettier-ignore /}
solidity
event ChainlinkCancelled(
    bytes32 indexed id
)

```

Emitted when `cancelChainlinkRequest` is called. Includes the request ID as an event topic.

Constants

LINKDIVISIBILITY

`LINKDIVISIBILITY` is a `uint256` constant to represent one whole unit of the LINK token (1000000000000000000). It can be used with another value to specify payment in an easy-to-read format, instead of hardcoding magic numbers.

```

{/ prettier-ignore /}
solidity
uint256 constant private ORACLEPAYMENT = 100 LINKDIVISIBILITY; // = 100 LINK

```

Modifiers

`recordChainlinkFulfillment`

`recordChainlinkFulfillment` is used on fulfillment callbacks to ensure that the caller and `requestId` are valid. This is the method equivalent of the method `validateChainlinkCallback`.

Either `validateChainlinkCallback` or `recordChainlinkFulfillment` should be used on all Chainlink callback functions to ensure that the sender and `requestId` are valid. They protect ChainlinkClient callbacks from being called by malicious callers. Do not call both of them, or your callback may revert before you can record the reported response.

```

{/ prettier-ignore /}
solidity

```

```
function myCallback(bytes32 requestId, uint256 price)
    public
    recordChainlinkFulfillment(requestId) // always validate callbacks
{
    currentPrice = price;
}
```

Chainlink.Request

```
{/ prettier-ignore /}
solidity
library Chainlink {
    struct Request {
        bytes32 id;
        address callbackAddress;
        bytes4 callbackFunctionId;
        uint256 nonce;
        Buffer.buffer buf;
    }
}
```

The Chainlink Request struct encapsulates all of the fields needed for a Chainlink request and its corresponding response callback.

The Chainlink protocol aims to be flexible and not restrict application developers. The Solidity Chainlink Request model is a great example of that. It is exceptionally flexible, given the limitations of Solidity. The request can contain an arbitrary amount of keys and values to be passed offchain to the oracles for each request. It does so by converting the parameters into CBOR, and then storing them in a buffer. This allows for any number of parameters all of different types to be encoded onchain.

The request's ID is generated by hashing the sender's address and the request's nonce. This scheme ensures that only the requester can generate their request ID, and no other contract can trigger a response from an oracle with that ID. New requests whose IDs match an unfulfilled request ID will not be accepted by the oracle.

<Aside type="caution" title="Intended for memory">

The Request object was intended to be stored in memory. If you have a reason to persist the struct in storage, it is recommended that you do so by copying each attribute over and explicitly copying the bytes in the buffer.

</Aside>

Attributes

Name	Description
-----	-----
:	:
-----	-----
id	Identifier for the request
callbackAddress	Address that the response will be sent to upon fulfillment
callbackFunctionId	Selector of the function on the callbackAddress that will be invoked with the response upon fulfillment
nonce	Used to generate the request ID
buf	Buffer that stores additional user defined parameters as

CBOR

|

Methods

Name	Description
:-----	

add	Add a string value to the run request parameters
addBytes	Add a bytes value to the run request parameters
addInt	Add an integer value to the run request parameters
addUint	Add an unsigned integer to the run request parameters
addStringArray	Add an array of strings as a value in the run request parameters
setBuffer	Directly set the CBOR of the run request parameters

add

```
{/ prettier-ignore /}  
solidity  
function add(  
    Request memory self,  
    string key,  
    string value  
)
```

Add a string value to the run request parameters. Commonly used for get with jobs using httpGet tasks.

```
{/ prettier-ignore /}  
solidity  
function requestEthereumPrice()  
    public  
{  
    Chainlink.Request memory req = buildChainlinkRequest(jobId, this,  
this.fulfill.selector);  
  
    req.add("get", "https://min-api.cryptocompare.com/data/price?  
fsym=ETH&tsyms=USD,EUR,JPY");  
  
    sendChainlinkRequest(req, 1 LINKDIVISIBILITY); // =1 LINK  
}
```

addBytes

```
{/ prettier-ignore /}  
solidity  
function addBytes(  
    Request memory self,  
    string key,  
    bytes value  
)
```

Add a CBOR bytes type value to the run request parameters.

```
{/ prettier-ignore /}
```

```

solidity
function requestEmojiPopularity(bytes unicode)
    public
{
    Chainlink.Request memory req = buildChainlinkRequest(jobId, this,
this.fulfill.selector);

    req.addBytes("emojiUnicode", unicode);

    sendChainlinkRequest(req, LINKDIVISIBILITY 1);
}

```

Note that this can also be used as a workaround to pass other data types like arrays or addresses. For instance, to add an address, one would first encode it using `abi.encode` then pass the result to `addBytes`:

```

{
  // prettier-ignore
}
solidity
Chainlink.Request memory req = buildChainlinkRequest(jobId, this,
this.fulfill.selector);

req.addBytes("address", abi.encode(msg.sender)); // msg.sender used in this
example. Replace it with your address

```

addInt

```

{
  // prettier-ignore
}
solidity
function addInt(
    Request memory self,
    string key,
    int256 value
)

```

Add a CBOR signed integer type value to the run request parameters. Commonly used with the `times` parameter of any job using a multiply task.

```

{
  // prettier-ignore
}
solidity
function requestPrice()
    public
{
    Chainlink.Request memory req = buildChainlinkRequest(jobId, this,
this.fulfill.selector);

    req.addInt("times", 100);

    sendChainlinkRequest(req, LINKDIVISIBILITY 1);
}

```

addUint

```

{
  // prettier-ignore
}
solidity
function addUint(
    Request memory self,
    string key,
    uint256 value
)

```

Add a CBOR unsigned integer type value to the run request parameters. Commonly used with the times parameter of any job using a multiply task.

```
{/ prettier-ignore /}
solidity
function requestPrice()
    public
{
    Chainlink.Request memory req = buildChainlinkRequest(jobId, this,
this.fulfill.selector);

    req.addUint("times", 100);

    sendChainlinkRequest(req, LINKDIVISIBILITY 1);
}
```

addStringArray

```
{/ prettier-ignore /}
solidity
function addStringArray(
    Request memory self,
    string key,
    string[] memory values
)
```

Add a CBOR array of strings to the run request parameters. Commonly used with the path parameter for any job including a jsonParse task.

```
{/ prettier-ignore /}
solidity
function requestPrice(string currency)
    public
{
    Chainlink.Request memory req = buildChainlinkRequest(JOBID, this,
this.myCallback.selector);
    string[] memory path = new string[](2);
    path[0] = currency;
    path[1] = "recent";

    // specify templated fields in a job specification
    req.addStringArray("path", path);

    sendChainlinkRequest(req, PAYMENT);
}
```

setBuffer

```
{/ prettier-ignore /}
solidity
function setBuffer(
    Request memory self,
    bytes data
)
```

Set the CBOR payload directly on the request object, avoiding the cost of encoding the parameters in CBOR. This can be helpful when reading the bytes from storage or having them passed in from offchain where they were pre-encoded.

```

{/ prettier-ignore /}
solidity
function requestPrice(bytes cbor)
    public
{
    Chainlink.Request memory req = buildChainlinkRequest(JOBID, this,
this.myCallback.selector);

    req.setBuffer(cbor);

    sendChainlinkRequest(req, PAYMENT);
}

```

<Aside type="caution" title="Be careful setting the request buffer directly">

Moving the CBOR encoding logic offchain can save some gas, but it also opens up the opportunity for people to encode parameters that not all parties agreed to. Be sure that whoever is permissioned to call setBuffer is trusted or auditable.

</Aside>

```
# find-oracle.mdx:
```

```

---
section: nodeOperator
date: Last Modified
title: "Find Existing Jobs"
isMdx: true
whatsnext: { "API Reference": "/any-api/api-reference/", "Testnet Oracles":
"/any-api/testnet-oracles/" }
---

```

```
import { Aside } from "@components"
```

This page explains how to find an existing Oracle Job to suit the needs of your API call.

<Aside type="note" title="Chainlink Functions">

Chainlink Functions provides your smart contracts access to trust-minimized compute infrastructure, allowing you to fetch data from APIs and perform custom computation. Read the Chainlink Functions documentation to learn more.

</Aside>

Introduction to Oracles

Oracles enable smart contracts to retrieve data from the outside world. Each oracle node can be configured to perform a wide range of tasks depending on the adapters it supports. For example, if your contract needs to make an HTTP GET request, it needs to use an oracle that supports the HTTP GET adapter.

Oracles jobs can be specialized even further by implementing the configuration using External Adapters. For example, an Oracle job could implement URL, parameters, and conversion to Solidity compatible data, to retrieve a very specific piece of data from a specific API endpoint. This process is demonstrated in Make an Existing Job Request.

Find a job

Community node operators

To find an Oracle Job that is pre-configured for your use case and available on the right network, join the Chainlink operator-requests discord channel to directly communicate with community node operators.

Alternatives on testnet

On testnet, several alternatives are provided:

- The Chainlink Development Relations team maintains Testnet Oracles that you can use to test implementations. If you don't find a suitable job for your needs, join Chainlink operator-requests discord channel or check the other alternatives below.
- Run your own testnet nodes as explained here. You must write your own jobs: To help you get started, each ANY API tutorial has a corresponding job attached to it.

getting-started.mdx:

```
---
section: nodeOperator
date: Last Modified
title: "Getting Started with Any API"
excerpt: "Calling APIs from Smart Contracts"
whatsnext:
  { "Learn how to bring data onchain and do offchain computation using Chainlink Functions": "/chainlink-functions" }
metadata:
  image: "/files/04b8e56-cl.png"
---
```

```
import { Aside, CodeSample } from "@components"
import { YouTube } from "@astro-community/astro-embed-youtube"
```

```
<Aside type="note" title="Chainlink Functions">
  Chainlink Functions provides your smart contracts access to trust-minimized
  compute infrastructure, allowing you to
  fetch data from APIs and perform custom computation. Read the Chainlink
  Functions
  documentation to learn more.
</Aside>
```

```
<Aside type="note" title="Requirements">
  This guide requires basic knowledge about smart contracts. If you are new to
  smart contract development, learn how to
  Deploy Your First Smart Contract before you begin.
</Aside>
```

```
<YouTube id="https://www.youtube.com/watch?v=ay4rXZhAefs" />
```

In this guide, you will learn how to request data from a public API in a smart contract. This includes understanding what Tasks and External adapters are and how Oracle Jobs use them. You will also learn how to find the Oracle Jobs and Tasks for your contract and how to request data from an Oracle Job.

How does the request and receive cycle work for API calls?

The request and receive cycle describes how a smart contract requests data from an oracle and receives the response in a separate transaction. If you need a refresher, check out the Basic Request Model.

For contracts that use Chainlink VRF, you request randomness from a VRF oracle and then await the response. The fulfillment function is already given to us

from the VRFConsumerBase contract, so oracles already know where to send the response to. However, with API calls, the contract itself defines which function it wants to receive the response to.

Before creating any code, you should understand how Oracle jobs can get data onchain.

What are jobs?

Chainlink nodes require Jobs to do anything useful. In the case of a Request and Receive job, the Direct Request job monitors the blockchain for a request from a smart contract. Once it catches a request, it runs the tasks (both core and external adapters) that the job is configured to run and eventually returns the response to the requesting contract.

What are tasks?

Each oracle job has a configured set of tasks it must complete when it is run. These tasks are split into two subcategories:

- Tasks - These are tasks that come built-in to each node. (examples: http, ethabiencode, etc).
- External Adapters - These are custom adapters built by node operators and community members, which perform specific tasks like calling a particular endpoint with a specific set of parameters (like authentication secrets that shouldn't be publicly visible).

Tasks

If a job needs to make a GET request to an API, find a specific unsigned integer field in a JSON response, then submit that back to the requesting contract, it would need a job containing the following Tasks:

- HTTP calls the API. the method must be set to GET.
- JSON Parse parses the JSON and extracts a value at a given keypath.
- Multiply multiplies the input by a multiplier. Used to remove the decimals.
- ETH ABI Encode converts the data to a bytes payload according to ETH ABI encoding.
- ETH Tx submits the transaction to the chain, completing the cycle.

The job specs example can be found [here](#).

Let's walk through a real example, where you will retrieve 24 volumes of the ETH/USD pair from the cryptocompare API.

1. HTTP calls the API and returns the body of an HTTP GET result for ETH/USD pair. Example:

```
{/ prettier-ignore /}
json
{"RAW":
  {"ETH":
    {"USD":
      {
        ...,
        "VOLUMEDAYTO":953806939.7194247,
        "VOLUME24HOUR":703946.0675653099,
        "VOLUME24HOURTO":1265826345.488568
        ...
      }
    }
  }
}
```

2. JSON Parse walks a specified path ("RAW,ETH,USD,VOLUME24HOUR") and returns the value found at that result. Example: 703946.0675653099

3. Multiply parses the input into a float and multiplies it by the 10^{18} . Example: 7039460675653099000000000

4. ETH ABI Encode formats the input into an integer and then converts it into Solidity's uint256 format. Example: 0xc618a1e4

5. ETH Tx takes the given input, places it into the data field of the transaction, signs a transaction, and broadcasts it to the network. Example: transaction result

Note: Some tasks accept parameters to be passed to them to inform them how to run. Example: JSON Parse accepts a path parameter which informs the task where to find the data in the JSON object.

Let's see what this looks like in a contract:

Contract example

```
<CodeSample src="/samples/APIRequests/APIConsumer.sol" />
```

Here is a breakdown of each component of this contract:

1. Constructor: This sets up the contract with the Oracle address, Job ID, and LINK fee that the oracle charges for the job.
2. requestVolumeData functions: This builds and sends a request - which includes the fulfillment functions selector - to the oracle. Notice how it adds the get, path and times parameters. These are read by the Tasks in the job to perform correctly. get is used by HTTP, path is used by JSON Parse and times is used by Multiply.
3. fulfill function: This is where the result is sent upon the Oracle Job's completion.

Note: The calling contract should own enough LINK to pay the fee, which by default is 0.1 LINK. You can use this tutorial to learn how to fund your contract.

This is an example of a basic HTTP GET request. However, it requires defining the API URL directly in the smart contract. This can, in fact, be extracted and configured on the Job level inside the Oracle node. You can follow the APIConsumer tutorial [here](#).

External adapters

If all the parameters are defined within the Oracle job, the only things a smart contract needs to define to consume are:

- JobId
- Oracle address
- LINK fee
- Fulfillment function

This will make your smart contract much more succinct. The requestVolumeData function from the code example above would look more like this:

```
{/ prettier-ignore /}  
solidity  
function requestVolumeData() public returns (bytes32 requestId) {  
    Chainlink.Request memory req = buildChainlinkRequest(jobId, address(this),  
this.fulfill.selector);  
  
    // Extra parameters don't need to be defined here because they are already
```

defined in the job

```
    return sendChainlinkRequest(req, fee);
}
```

You can follow a full Existing Job Tutorial [here](#).
More on External Adapters can be found [here](#).

Further reading

To learn more about connecting smart contracts to external APIs, read our blog posts:

- [Connect a Smart Contract to the Twitter API](#)
- [Connect a Tesla Vehicle API to a Smart Contract](#)
- [OAuth and API Authentication in Smart Contracts](#)

introduction.mdx:

```
---
section: nodeOperator
date: Last Modified
title: "Chainlink Any API Documentation"
isMdx: true
whatsnext:
  {
    "Make a GET Request": "/any-api/get-request/introduction/",
    "API Reference": "/any-api/api-reference/",
    "Find Existing Jobs": "/any-api/find-oracle/",
    "Testnet Oracles": "/any-api/testnet-oracles/",
  }
metadata:
  title: "Request and Receive API Data with Chainlink"
  description: "Chainlink provides your smart contract with access to any
external API. Learn how to integration any API into your smart contract."
  image: "/files/bc12c34-link.png"
---

import AnyApiCallout from "@features/any-api/common/AnyApiCallout.astro"
import { Aside } from "@components"

<Aside type="note" title="Talk to an expert">
  <a href="https://chain.link/contact?refid=AnyAPI">Contact us</a> to talk to an
expert about using Chainlink Any API
  to get your data on chain.
</Aside>

<Aside type="note" title="Chainlink Functions">
  Chainlink Functions provides your smart contracts access to trust-minimized
compute infrastructure, allowing you to
  fetch data from APIs and perform custom computation. Read the Chainlink
Functions
  documentation to learn more.
</Aside>
```

Connecting to any API with Chainlink enables your contracts to access to any external data source through our decentralized oracle network. We understand making smart contracts compatible with offchain data adds to the complexity of building smart contracts. We created a framework with minimal requirements, yet unbounded flexibility, so developers can focus more on the functionality of smart contracts rather than what feeds them. Chainlink's decentralized oracle network provides smart contracts with the ability to push and pull data,

Whether your contract requires sports results, the latest weather, or any other publicly available data, the Chainlink contract library provides the tools required for your contract to consume it.

```
<Aside type="note" title="Note on Price Feed Data">
```

</Aside>

Outlined below are multiple ways developers can connect smart contracts to offchain data feeds. Click a request type to learn more about it:

To understand different use cases for using any API, refer to [Other Tutorials](#).

```

---
section: nodeOperator
date: Last Modified

```



```

-----
|
| GET>bytes: <br />HTTP GET to any public API <br />parse the response <br
/>return arbitrary-length raw byte data bytes. <br />The job specs can be found
here | Http<br />JsonParse<br
/>Ethabiencode
| 7da2702f37fd48e5b1b9a5715e3509b6 | <ul><li>get: string</li><li>path: JSONPath
expression with comma(,) delimited string</li></ul> |
| GET>uint256: <br />HTTP GET to any public API <br />parse the response <br
/>multiply the result by a multiplier <br />return an unsigned integer uint256 .
<br /> The job specs can be found here | Http<br />JsonParse<br />Multiply<br
/>Ethabiencode | ca98366cc7314957b8c012c72f05aeeb | <ul><li>get:
string</li><li>path: JSONPath expression with comma(,) delimited
string</li><li>times: int</li></ul> |
| GET>int256: <br />HTTP GET to any public API <br />parse the response <br
/>multiply the result by a multiplier <br />return a signed integer int256.
<br /> The job specs can be found here | Http<br />JsonParse<br
/>Multiply<br />Ethabiencode | fcf4140d696d44b687012232948bdd5d | <ul><li>get:
string</li><li>path: JSONPath expression with comma(,) delimited
string</li><li>times: int</li></ul> |
| GET>bool: <br />HTTP GET to any public API <br />parse the response <br
/>return a boolean bool. <br /> The job specs can be found here
| Http<br />JsonParse<br />Ethabiencode
| c1c5e92880894eb6b27d3cae19670aa3 | <ul><li>get: string</li><li>path: JSONPath
expression with comma(,) delimited string</li></ul> |
| GET>string: <br />HTTP GET to any public API <br />parse the response <br
/>return a sequence of characters string. <br /> The job specs can be found here
| Http<br />JsonParse<br />Ethabiencode
| 7d80a6386ef543a3abb52817f6707e3b | <ul><li>get: string</li><li>path: JSONPath
expression with comma(,) delimited string</li></ul> |
</div>

```

Examples

Get > bytes

A full example can be found here.

Request method

```

{/ prettier-ignore /}
solidity
function request() public {
    Chainlink.Request memory req =
    buildChainlinkRequest('7da2702f37fd48e5b1b9a5715e3509b6', address(this),
    this.fulfill.selector);
    req.add(
        'get',
        'https://ipfs.io/ipfs/QmZgsvrA1o1C8BGCrX6mHTqR1Ui1XqbCrtbMVrRLHtuPVD?
filename=big-api-response.json'
    );
    req.add('path', 'image');
    sendChainlinkRequest(req, (1 LINKDIVISIBILITY) / 10); // 0,11018 LINK
}

```

Callback method

```

{/ prettier-ignore /}
solidity
bytes public data;
string public imageUrl;
function fulfill(bytes32 requestId, bytes memory bytesData) public

```

```
recordChainlinkFulfillment(requestId) {
    data = bytesData;
    imageUrl = string(data);
}
```

Get > uint256

A full example can be found [here](#).

Request method

```
{/ prettier-ignore /}
solidity
function request() public {
    Chainlink.Request memory req =
    buildChainlinkRequest('ca98366cc7314957b8c012c72f05aeeb', address(this),
    this.fulfill.selector);
    req.add(
        'get',
        'https://min-api.cryptocompare.com/data/pricemultifull?
fsyms=ETH&tsyms=USD'
    );
    req.add('path', 'RAW,ETH,USD,VOLUME24HOUR');
    req.addInt('times', 1018); // Multiply by times value to remove decimals.
    Parameter required so pass '1' if the number returned doesn't have decimals
    sendChainlinkRequest(req, (1 LINKDIVISIBILITY) / 10); // 0,11018 LINK
}
```

Callback method

```
{/ prettier-ignore /}
solidity
uint256 public volume;
function fulfill(bytes32 requestId, uint256 volume) public
recordChainlinkFulfillment(requestId) {
    volume = volume;
}
```

Get > int256

Request method

```
{/ prettier-ignore /}
solidity
function request() public {
    Chainlink.Request memory req =
    buildChainlinkRequest('fcf4140d696d44b687012232948bdd5d', address(this),
    this.fulfill.selector);
    req.add(
        'get',
        'https://min-api.cryptocompare.com/data/pricemultifull?
fsyms=ETH&tsyms=USD'
    );
    req.add('path', 'RAW,ETH,USD,VOLUME24HOUR');
    req.addInt('times', 1018); // Multiply by times value to remove decimals.
    Parameter required so pass '1' if the number returned doesn't have decimals
    sendChainlinkRequest(req, (1 LINKDIVISIBILITY) / 10); // 0,11018 LINK
}
```

Callback method


```

{/ prettier-ignore /}
solidity
int256 public volume;
function fulfill(bytes32 requestId, int256 volume) public
recordChainlinkFulfillment(requestId) {
    volume = volume;
}

```

Get > bool

Request method

```

{/ prettier-ignore /}
solidity
function request() public {
    Chainlink.Request memory req =
    buildChainlinkRequest('c1c5e92880894eb6b27d3cae19670aa3', address(this),
    this.fulfill.selector);
    req.add(
        'get',

'https://app.proofi.com/api/verify/eip155/0xCB5085214B6318aF3dd0FBbb5E74fbF6bf33
2151?contract=0x2f7f7E44ca1e2Ca1A54db4222cF97ab47EE026F1'
    );
    req.add('path', 'approved');
    sendChainlinkRequest(req, (1 LINKDIVISIBILITY) / 10); // 0,11018 LINK
}

```

Callback method

```

{/ prettier-ignore /}
solidity
bool public approved;
function fulfill(bytes32 requestId, bool approved) public
recordChainlinkFulfillment(requestId) {
    approved = approved;
}

```

Get > string

A full example can be found [here](#).

Request method

```

{/ prettier-ignore /}
solidity
function request() public {
    Chainlink.Request memory req =
    buildChainlinkRequest('7d80a6386ef543a3abb52817f6707e3b', address(this),
    this.fulfill.selector);
    req.add(
        'get',
        'https://api.coingecko.com/api/v3/coins/markets?vscurrency=usd&perpage=10'
    );
    req.add('path', '0,id');
    sendChainlinkRequest(req, (1 LINKDIVISIBILITY) / 10); // 0,11018 LINK
}

```

Callback method

```

{/ prettier-ignore /}
solidity
string public id;
function fulfill(bytes32 requestId, string memory id) public
recordChainlinkFulfillment(requestId) {
    id = id;
}

```

introduction.mdx:

```

---
section: nodeOperator
date: Last Modified
title: "Make a GET Request"
isMdx: true
whatsnext:
  {
    "Single Word Response": "/any-api/get-request/examples/single-word-
response/",
    "Multi-Variable Responses": "/any-api/get-request/examples/multi-variable-
responses/",
    "Fetch data from an Array": "/any-api/get-request/examples/array-response/",
    "Large Responses": "/any-api/get-request/examples/large-responses/",
    "Make an Existing Job Request": "/any-api/get-request/examples/existing-job-
request/",
    "API Reference": "/any-api/api-reference/",
    "Testnet Oracles": "/any-api/testnet-oracles/",
  }
metadata:
  title: "Make a GET Request"
  description: "Learn how to make a GET request to an API from a smart contract,
using Chainlink."
  image: "/files/930cbb7-link.png"
---

```

```

import { Aside } from "@components"
import AnyApiCallout from "@features/any-api/common/AnyApiCallout.astro"

```

This series of guides explains how to make HTTP GET requests to external APIs from smart contracts, using Chainlink's Request & Receive Data cycle.

```

<Aside type="note" title="Chainlink Functions">
  Chainlink Functions provides your smart contracts access to trust-minimized
compute infrastructure, allowing you to
  fetch data from APIs and perform custom computation. Read the Chainlink
Functions
  documentation to learn more.
</Aside>

```

```

<AnyApiCallout callout="common" />

```

Examples

Single Word Response

This guide explains how to make an HTTP GET request and parse the json response to retrieve the value of one single attribute.

Multi-Variable Responses

This guide explains how to make an HTTP GET request and parse the json response

to retrieve the values of multiple attributes.

Fetch data from an Array

This guide explains how to make an HTTP GET request that returns a json array and parse it to retrieve the target element's value.

Large Responses

This guide explains how to make an HTTP Get request that returns a json containing an arbitrary-length raw byte data and parse it to return the data as bytes data type.

Make an Existing Job Request

This guide explains how to call a job that leverages External adapters and returns the relevant data to the smart contract. This allows building succinct smart contracts that do not need to comprehend the URL or the response format of the target API.

```
# array-response.mdx:
```

```
---
section: nodeOperator
date: Last Modified
title: "Array Response"
isMdx: true
whatsnext:
  {
    "Large Responses": "/any-api/get-request/examples/large-responses/",
    "Make an Existing Job Request": "/any-api/get-request/examples/existing-job-request/",
    "API Reference": "/any-api/api-reference/",
    "Testnet Oracles": "/any-api/testnet-oracles/",
  }
---
```

```
import { Aside, CodeSample } from "@components"
import AnyApiCallout from "@features/any-api/common/AnyApiCallout.astro"
```

This guide explains how to make an HTTP GET request to an external API, that returns a json array, from a smart contract, using Chainlink's Request & Receive Data cycle and then receive the needed data from the array.

```
<Aside type="note" title="Chainlink Functions">
  Chainlink Functions provides your smart contracts access to trust-minimized
  compute infrastructure, allowing you to
  fetch data from APIs and perform custom computation. Read the Chainlink
  Functions
  documentation to learn more.
</Aside>
```

```
<AnyApiCallout callout="prerequisites" />
```

Example

This example shows how to:

- Call an API that returns a JSON array.
- Fetch a specific information from the response.

Coingecko GET /coins/markets/ API returns a list of coins and their market data such as price, market cap, and volume. To check the response, you can directly

paste the following URL in your browser
<https://api.coingecko.com/api/v3/coins/markets?vscurrency=usd&order=marketcapdesc&perpage=100&page=1&sparkline=false> or run this command in your terminal:

```
bash
curl -X 'GET' \
  'https://api.coingecko.com/api/v3/coins/markets?vscurrency=usd&order=marketcapdesc&perpage=100&page=1&sparkline=false' \
  -H 'accept: application/json'
```

The response should be similar to the following:

```
{/ prettier-ignore /}
json
[
  {
    "id": "bitcoin",
    "symbol": "btc",
    "name": "Bitcoin",
    "image": "https://assets.coingecko.com/coins/images/1/large/bitcoin.png?1547033579",
    "currentprice": 42097,
    "marketcap": 802478449872,
    ...
  },
  {
    ...
  }
]
```

Fetch the id of the first element. To consume an API, your contract must import ChainlinkClient.sol. This contract exposes a struct named Chainlink.Request, which your contract can use to build the API request. The request must include the following parameters:

- Link token address
- Oracle address
- Job id
- Request fee
- Task parameters
- Callback function signature

<Aside type="caution" title="Note on Funding Contracts">

Making a GET request will fail unless your deployed contract has enough LINK to pay for it. Learn how to Acquire testnet LINK and Fund your contract.

</Aside>

<CodeSample src="samples/APIRequests/FetchFromArray.sol" />

To use this contract:

1. Open the contract in Remix.

1. Compile and deploy the contract using the Injected Provider environment. The contract includes all the configuration variables for the Sepolia testnet. Make sure your wallet is set to use Sepolia. The constructor sets the following parameters:

- The Chainlink Token address for Sepolia by calling the setChainlinkToken

function.

- The Oracle contract address for Sepolia by calling the setChainlinkOracle function.

- The jobId: A specific job for the oracle node to run. In this case, the id is a string data type, so you must call a job that calls an API and returns a string. We will be using a generic GET>string job that can be found here.

1. Fund your contract with 0.1 LINK. To learn how to send LINK to contracts, read the Fund Your Contracts page.

1. Call the id function to confirm that the id state variable is not set.

1. Run the requestFirstId function. This builds the Chainlink.Request using the correct parameters. The req.add("path", "0,id") request parameter tells the oracle node to fetch the id at index 0 of the array returned by the GET request. It uses JSONPath expression with comma , delimited string for nested objects, for example: '0,id'.

1. After few seconds, call the id function. You should get a non-empty response: bitcoin

```
<AnyApiCallout callout="common" />
```

```
# existing-job-request.mdx:
```

```
---
section: nodeOperator
date: Last Modified
title: "Existing Job Request"
isMdx: true
whatsnext:
  {
    "Find Existing Jobs": "/any-api/find-oracle/",
    "API Reference": "/any-api/api-reference/",
    "Testnet Oracles": "/any-api/testnet-oracles/",
  }
metadata:
  title: "Make an Existing Job Request"
  description: "Learn how to use existing Chainlink external adapters to make calls to APIs from smart contracts."
  image: "/files/OpenGraphV3.png"
---
```

```
import { Aside, CodeSample } from "@components"
import AnyApiCallout from "@features/any-api/common/AnyApiCallout.astro"
```

Using an existing Oracle Job makes your smart contract code more succinct. This page explains how to retrieve the gas price from an existing Chainlink job that calls etherscan gas tracker API.

```
<Aside type="note" title="Chainlink Functions">
```

Chainlink Functions provides your smart contracts access to trust-minimized compute infrastructure, allowing you to fetch data from APIs and perform custom computation. Read the Chainlink Functions documentation to learn more.

```
</Aside>
```

```
<AnyApiCallout callout="prerequisites" />
```

Example

In Single Word Response Example, the example contract code declared which URL to

use, where to find the data in the response, and how to convert it so that it can be represented onchain.

This example uses an existing job that is pre-configured to make requests to get the gas price. Using specialized jobs makes your contracts succinct and more simple.

Etherscan gas oracle returns the current Safe, Proposed and Fast gas prices. To check the response, you can directly paste the following URL in your browser <https://api.etherscan.io/api?module=gastracker&action=gasoracle&apikey=YourApiKeyToken> or run this command in your terminal:

```
bash
curl -X 'GET' \
  'https://api.etherscan.io/api?
module=gastracker&action=gasoracle&apikey=YourApiKeyToken' \
  -H 'accept: application/json'
```

The response should be similar to the following:

```
json
{
  "status": "1",
  "result": {
    "LastBlock": "14653286",
    "SafeGasPrice": "33",
    "ProposeGasPrice": "33",
    "FastGasPrice": "35",
    "suggestBaseFee": "32.570418457",
    "gasUsedRatio":
"0.366502543599508,0.15439818258491,0.9729006,0.4925609,0.999657066666667"
  }
}
```

For this example, we created a job that leverages the EtherScan External Adapter to fetch the SafeGasPrice , ProposeGasPrice and FastGasPrice. You can learn more about External Adapters [here](#).

To consume an API, your contract must import ChainlinkClient.sol. This contract exposes a struct named Chainlink.Request, which your contract can use to build the API request. The request must include the following parameters:

- Link token address
- Oracle address
- Job id
- Request fee
- Task parameters
- Callback function signature

<Aside type="caution" title="Note on Funding Contracts">

Making a GET request will fail unless your deployed contract has enough LINK to pay for it. Learn how to Acquire testnet LINK and Fund your contract.

</Aside>

<CodeSample src="samples/APIRequests/GetGasPrice.sol" />

To use this contract:

1. Open the contract in Remix.

1. Compile and deploy the contract using the Injected Provider environment. The contract includes all the configuration variables for the Sepolia testnet. Make sure your wallet is set to use Sepolia. The constructor sets the following parameters:

- The Chainlink Token address for Sepolia by calling the `setChainlinkToken` function.
- The Oracle contract address for Sepolia by calling the `setChainlinkOracle` function.
- The `jobId`: A specific job for the oracle node to run. In this case, the job is very specific to the use case as it returns the gas prices. You can find the job spec for the Chainlink node [here](#).

1. Fund your contract with 0.1 LINK. To learn how to send LINK to contracts, read the [Fund Your Contracts](#) page.

1. Call the `gasPriceFast`, `gasPriceAverage` and `gasPriceSafe` functions to confirm that the `gasPriceFast`, `gasPriceAverage` and `gasPriceSafe` state variables are equal to zero.

1. Run the `requestGasPrice` function. This builds the `Chainlink.Request`. Note how succinct the request is.

1. After few seconds, call the `gasPriceFast`, `gasPriceAverage` and `gasPriceSafe` functions. You should get a non-zero responses.

```
<AnyApiCallout callout="common" />
```

```
# large-responses.mdx:
```

```
---
section: nodeOperator
date: Last Modified
title: "Large Responses"
isMdx: true
whatsnext:
  {
    "Make an Existing Job Request": "/any-api/get-request/examples/existing-job-request/",
    "API Reference": "/any-api/api-reference/",
    "Testnet Oracles": "/any-api/testnet-oracles/",
  }
---
```

```
import { Aside, CodeSample } from "@components"
import AnyApiCallout from "@features/any-api/common/AnyApiCallout.astro"
```

This guide explains how to make an HTTP GET request to an external API from a smart contract, using Chainlink's Request & Receive Data cycle and then receive large responses.

```
<Aside type="note" title="Chainlink Functions">
  Chainlink Functions provides your smart contracts access to trust-minimized
  compute infrastructure, allowing you to
  fetch data from APIs and perform custom computation. Read the Chainlink
  Functions
  documentation to learn more.
</Aside>
```

```
<AnyApiCallout callout="prerequisites" />
```

Example

This example shows how to:

- Call an API and fetch the response that is an arbitrary-length raw byte data.

IPFS is a decentralized file system for storing and accessing files, websites, applications, and data. For this example, we stored in IPFS a JSON file that contains arbitrary-length raw byte data. To check the response, directly paste the following URL in your browser:

<https://ipfs.io/ipfs/QmZgsvrA1o1C8BGCrx6mHTqR1Ui1XqbCrtbMvRLHtuPVD?filename=big-api-response.json> Alternatively, run the following command in your terminal:

```
bash
curl -X 'GET' \
  'https://ipfs.io/ipfs/QmZgsvrA1o1C8BGCrx6mHTqR1Ui1XqbCrtbMvRLHtuPVD?filename=big-api-response.json' \
  -H 'accept: application/json'
```

The response should be similar to the following:

```
json
{
  "image":
  "0x68747470733a2f2f697066732e696f2f697066732f516d5358416257356b716e3259777435444c336857354d736a654b4a4839724c654c6b51733362527579547871313f66696c656e616d653d73756e2d636861696e6c696e6b2e676966"
}
```

Fetch the value of image. To consume an API, your contract must import ChainlinkClient.sol. This contract exposes a struct named Chainlink.Request, which your contract can use to build the API request. The request must include the following parameters:

- Link token address
- Oracle address
- Job id
- Request fee
- Task parameters
- Callback function signature

<Aside type="caution" title="Note on Funding Contracts">

Making a GET request will fail unless your deployed contract has enough LINK to pay for it. Learn how to Acquire testnet LINK and Fund your contract.

</Aside>

<CodeSample src="samples/APIRequests/GenericBigWord.sol" />

To use this contract:

1. Open the contract in Remix.

1. Compile and deploy the contract using the Injected Provider environment. The contract includes all the configuration variables for the Sepolia testnet. Make sure your wallet is set to use Sepolia. The constructor sets the following parameters:

- The Chainlink Token address for Sepolia by calling the setChainlinkToken function.
- The Oracle contract address for Sepolia by calling the setChainlinkOracle function.

- The jobId: A specific job for the oracle node to run. In this case, the data is a bytes data type, so you must call a job that calls an API and returns bytes. We will be using a generic GET>bytes job that can be found here.

1. Fund your contract with 0.1 LINK. To learn how to send LINK to contracts, read the Fund Your Contracts page.

1. Call the data and imageUrl functions to confirm that the data and imageUrl state variables are not set.

1. Run the requestBytes function. This builds the Chainlink.Request using the correct parameters:

- The req.add("get", "URL") request parameter provides the oracle node with the url where to fetch the response.
- The req.add('path', 'image') request parameter tells the oracle node how to parse the response.

1. After few seconds, call the data and imageUrl functions. You should get non-empty responses.

```
<AnyApiCallout callout="common" />
```

```
# multi-variable-responses.mdx:
```

```
---
section: nodeOperator
date: Last Modified
title: "Multi-Variable Responses"
isMdx: true
whatsnext:
  {
    "Fetch data from an Array": "/any-api/get-request/examples/array-response/",
    "Large Responses": "/any-api/get-request/examples/large-responses/",
    "Make an Existing Job Request": "/any-api/get-request/examples/existing-job-request/",
    "API Reference": "/any-api/api-reference/",
    "Testnet Oracles": "/any-api/testnet-oracles/",
  }
---
```

```
import { Aside, CodeSample } from "@components"
import AnyApiCallout from "@features/any-api/common/AnyApiCallout.astro"
```

This guide explains how to make an HTTP GET request to an external API from a smart contract, using Chainlink's Request & Receive Data cycle and then receive multiple responses.

This is known as multi-variable or multi-word responses.

```
<Aside type="note" title="Chainlink Functions">
  Chainlink Functions provides your smart contracts access to trust-minimized
  compute infrastructure, allowing you to
  fetch data from APIs and perform custom computation. Read the Chainlink
  Functions
  documentation to learn more.
</Aside>
```

```
<AnyApiCallout callout="prerequisites" />
```

Example

This example shows how to:

- Fetch several responses in one single call.

Cryptocompare GET /data/price/ API returns the current price of any cryptocurrency in any other currency that you need. To check the response, you can directly paste the following URL in your browser <https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=BTC> or run this command in your terminal:

```
bash
curl -X 'GET' \
  'https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=BTC' \
  -H 'accept: application/json'
```

The response should be similar to the following:

```
json
{
  "BTC": 0.07297
}
```

The request above shows how to get the price of ETH against BTC. Now let say we want the price of ETH against several currencies: BTC, USD, and EUR. Our contract will have to support receiving multiple responses.

To consume an API with multiple responses, your contract should inherit from ChainlinkClient. This contract exposes a struct called Chainlink.Request, which your contract should use to build the API request. The request should include the following parameters:

- Link token address
- Oracle address
- Job id
- Request fee
- Task parameters
- Callback function signature

<Aside type="caution" title="Note on Funding Contracts">

Making a GET request will fail unless your deployed contract has enough LINK to pay for it. Learn how to Acquire testnet LINK and Fund your contract.

</Aside>

Assume that a user wants to obtain the ETH price quoted against three different currencies: BTC , USD and EUR. If they use only a single-word job, it would require three different requests. For a comparison, see the Single Word Response example. To make these requests more efficient, use multi-word responses to do it all in a single request as shown in the following example:

<CodeSample src="samples/APIRequests/MultiWordConsumer.sol" />

To use this contract:

1. Open the contract in Remix.

1. Compile and deploy the contract using the Injected Provider environment. The contract includes all the configuration variables for the Sepolia testnet. Make sure your wallet is set to use Sepolia. The constructor sets the following parameters:

- The Chainlink Token address for Sepolia by calling the setChainlinkToken function.
- The Oracle contract address for Sepolia by calling the setChainlinkOracle

function.

- The jobId: A specific job for the oracle node to run. In this case, you must call a job that is specifically configured to return ETH price against BTC, USD and EUR. You can find the job spec for the Chainlink node [here](#).

1. Fund your contract with 0.1 LINK. To learn how to send LINK to contracts, read the [Fund Your Contracts](#) page.

1. Call the btc, usd , and eur functions to confirm that the respective btc, usd , and eur state variables are equal to zero.

1. Run the requestMultipleParameters function. This builds the Chainlink.Request using the correct parameters:

- The req.add("urlBTC", "CRYPTOCOMPAREETHBTCURL") request parameter provides the oracle node with the url where to fetch the ETH-BTC price. Same logic for req.add("urlEUR", "CRYPTOCOMPAREETHBTCURL") and req.add("urlUSD", "CRYPTOCOMPAREETHBTCURL").
- The req.add('pathBTC', 'BTC') request parameter tells the oracle node where to fetch the ETH-BTC price in the json response. Same logic for req.add('pathUSD', 'EUR') and req.add('pathEUR', 'USD').

Because you provide the URLs and paths, the MultiWordConsumer in the example can call any public API as long as the URLs and paths are correct.

1. After few seconds, call the btc, usd , and eur functions. You should get a non-zero responses.

The job spec for the Chainlink node in this example can be found [here](#).

```
<AnyApiCallout callout="common" />
```

```
# single-word-response.mdx:
```

```
---
section: nodeOperator
date: Last Modified
title: Single Word Response
isMdx: true
whatsnext:
  {
    "Multi-Variable Responses": "/any-api/get-request/examples/multi-variable-responses/",
    "Fetch data from an Array": "/any-api/get-request/examples/array-response/",
    "Large Responses": "/any-api/get-request/examples/large-responses/",
    "Make an Existing Job Request": "/any-api/get-request/examples/existing-job-request/",
    "API Reference": "/any-api/api-reference/",
    "Testnet Oracles": "/any-api/testnet-oracles/",
  }
metadata:
  title: "Single Word Response"
  description: "Learn how to make a GET request to an API from a smart contract, using Chainlink."
  image: "/files/930cbb7-link.png"
---
```

```
import { Aside, CodeSample } from "@components"
import AnyApiCallout from "@features/any-api/common/AnyApiCallout.astro"
```

This guide explains how to make an HTTP GET request to an external API from a smart contract using Chainlink's Request & Receive Data cycle and receive a single response.

```
<Aside type="note" title="Chainlink Functions">
```

Chainlink Functions provides your smart contracts access to trust-minimized compute infrastructure, allowing you to fetch data from APIs and perform custom computation. Read the Chainlink Functions documentation to learn more.

<AnyApiCallout callout="prerequisites" />

Example

This example shows how to:

- Fetch a single word response in a single call.

The Cryptocompare GET /data/pricemultifull API returns the current trading info (price, vol, open, high, low) of any list of cryptocurrencies in any other currency that you need. To check the response, you can directly paste the following URL in your browser <https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD> or run this command in your terminal:

```
bash
curl -X 'GET' \
  'https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD' \
  -H 'accept: application/json'
```

The response should be similar to the following example:

```
{/ prettier-ignore /}
json
{
  "RAW": {
    "ETH": {
      "USD": {
        "TYPE": "5",
        "MARKET": "CCCAGG",
        "FROMSYMBOL": "ETH",
        "TOSYMBOL": "USD",
        "FLAGS": "2049",
        "PRICE": 2867.04,
        "LASTUPDATE": 1650896942,
        "MEDIAN": 2866.2,
        "LASTVOLUME": 0.16533939,
        "LASTVOLUMETO": 474.375243849,
        "LASTTRADEID": "1072154517",
        "VOLUMEDAY": 195241.78281014622,
        "VOLUMEDAYTO": 556240560.4621655,
        "VOLUME24HOUR": 236248.94641103,
        ...
      }
    }
  }
}
```

To consume an API with multiple responses, your contract must import ChainlinkClient. This contract exposes a struct called Chainlink.Request, which your contract should use to build the API request. The request should include the following parameters:

- Link token address
- Oracle address
- Job id
- Request fee
- Task parameters

- Callback function signature

<Aside type="caution" title="Note on Funding Contracts">

Making a GET request will fail unless your deployed contract has enough LINK to pay for it. Learn how to Acquire testnet LINK and Fund your contract.

</Aside>

Assume that a user wants to call the API above and retrieve only the 24h ETH trading volume from the response.

<CodeSample src="samples/APIRequests/APIConsumer.sol" />

To use this contract:

1. Open the contract in Remix.

1. Compile and deploy the contract using the Injected Provider environment. The contract includes all the configuration variables for the Sepolia testnet. Make sure your wallet is set to use Sepolia. The constructor sets the following parameters:

- The Chainlink Token address for Sepolia by calling the setChainlinkToken function.
- The Oracle contract address for Sepolia by calling the setChainlinkOracle function.
- The jobId: A specific job for the oracle node to run. In this case, you must call a job that is configured to call a public API, parse a number from the response and remove any decimals from it. We are going to use a generic GET>uint256 job that can be found here.

1. Fund your contract with 0.1 LINK. To learn how to send LINK to contracts, read the Fund Your Contracts page.

1. Call the volume function to confirm that the volume state variable is equal to zero.

1. Run the requestVolumeData function. This builds the Chainlink.Request using the correct parameters:

- The req.add("get", "CRYPTOCOMPAREURL") request parameter provides the oracle node with the URL from which to fetch ETH-USD trading info.
- The req.add('path', 'RAW,ETH,USD,VOLUME24HOUR') request parameter tells the oracle node where to fetch the 24h ETH volume in the json response. It uses a JSONPath expression with comma(,) delimited string for nested objects. For example: 'RAW,ETH,USD,VOLUME24HOUR'.
- The req.addInt('times', timesAmount) request parameter provides the oracle node with the multiplier timesAmount by which the fetched volume is multiplied. Use this to remove any decimals from the volume. Note: The times parameter is mandatory. If the API that you call returns a number without any decimals then provide 1as timesAmount.

The APIConsumer in the example above is flexible enough to call any public API as long as the URL in get, path, and timesAmount are correct.

1. After few seconds, call the volume function. You should get a non-zero response.

<AnyApiCallout callout="common" />

architecture-decentralized-model.mdx:

```
section: global
date: Last Modified
title: "Decentralized Data Model"
whatsnext:
  {
    "Using Data Feeds": "/data-feeds/price-feeds/",
    "Offchain Reporting": "/architecture-overview/off-chain-reporting/",
  }
metadata:
  title: "Chainlink Decentralised Data Model"
  description: "This page describes the decentralized architecture which enables Chainlink to aggregate data from multiple independent node operators."
  image: "/files/OpenGraphV3.png"
---
```

```
import { ClickToZoom } from "@components"
```

This page describes how data aggregation is applied to produce Chainlink Data Feeds and provides more insight as to how Data Feeds are updated.

Data aggregation

Each data feed is updated by multiple, independent Chainlink oracle operators. The AccessControlledOffchainAggregator aggregates the data onchain.

Offchain Reporting (OCR) further enhances the aggregation process. To learn more about OCR and how it works, see the Offchain Reporting page.

```
<ClickToZoom src="/images/contract-devs/price-aggr.png" />
```

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 aggregator contract must receive responses from a minimum number of oracles or the latest answer will not be updated. You can see the minimum number of oracles for the corresponding feed at data.chain.link.

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.

Components of a Decentralized Oracle Network

Data Feeds are an example of a decentralized oracle network, and include the following components:

- A consumer contract
- A proxy contract
- An aggregator contract

To learn how to create a consumer contract that uses an existing data feed, read the Using Data Feeds documentation.

Consumer

A Consumer contract is any contract that uses Chainlink Data Feeds to consume aggregated data. Consumer contracts must reference the correct `AggregatorV3Interface` contract and call one of the exposed functions.

```
{/ prettier-ignore /}  
solidity  
...  
AggregatorV3Interface feed = AggregatorV3Interface(address);  
return feed.latestRoundData();
```

Offchain applications can also consume data feeds. See the Javascript and Python example code on the Using Data Feeds page to learn more.

Proxy

Proxy contracts are onchain proxies that point to the aggregator for a particular data feed. Using proxies enables the underlying aggregator to be upgraded without any service interruption to consuming contracts.

Proxy contracts can vary from one data feed to another, but the `EACAggregatorProxy.sol` contract on Github is a common example.

Aggregator

An aggregator is the contract that receives periodic data updates from the oracle network. Aggregators store aggregated data onchain so that consumers can retrieve it and act upon it within the same transaction.

You can access this data using the Data Feed address and the `AggregatorV3Interface` contract.

Aggregators receive updates from the oracle network only when the Deviation Threshold or Heartbeat Threshold triggers an update during an aggregation round. 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 onchain 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.

architecture-overview.mdx:

```
---  
section: global  
date: Last Modified  
title: "Data Feeds Architecture"  
whatsnext:  
  {  
    "Basic Request Model": "/architecture-overview/architecture-request-model/",  
    "Decentralized Data Model": "/architecture-overview/architecture-decentralized-model/",  
    "Offchain Reporting": "/architecture-overview/off-chain-reporting/",  
  }  
metadata:  
  title: "Data Feeds Architecture"  
---
```

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 describes the onchain architecture of requesting data from a single oracle source.

To learn how to make a GET request using a single oracle, see [Make a GET Request](#).

Decentralized data model

For a more robust and trustworthy answer, you can aggregate data from many oracles. With onchain 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 describes how data is aggregated, and how consumer contracts can retrieve this data.

Offchain reporting

Offchain Reporting (OCR) is an improvement on the decentralization and scalability of Chainlink networks. With our Offchain Reporting aggregators, all nodes communicate using a peer to peer network. During the communication process, a lightweight consensus algorithm runs where each node reports its price observation and signs it. A single aggregate transaction is then transmitted, which saves a significant amount of gas.

To learn more about OCR and how it works, see the [Offchain Reporting page](#).

```
# architecture-request-model.mdx:
```

```
---
section: global
date: Last Modified
title: "Basic Request Model"
whatsnext:
  {
    "Make a GET Request": "/any-api/get-request/introduction/",
    "Decentralized Data Model": "/architecture-overview/architecture-
decentralized-model/",
    "Offchain Reporting": "/architecture-overview/off-chain-reporting/",
  }
metadata:
  title: "Chainlink Basic Request Model"
---
```

```
import { ClickToZoom } from "@components"
```

Contracts overview

All source code is open source and available in the Chainlink Github repository.

```
<ClickToZoom src="/files/881ade6-SimpleArchitectureDiagram1V1.png" />
```

ChainlinkClient

ChainlinkClient 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.

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`.

To build your own client contract using ChainlinkClient, see [Introduction to Using Any API](#), or view the ChainlinkClient API Reference for the ChainlinkClient contract.

LINK Token

LINK is an ERC-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.

Learn more about ERC-677 and the LINK token.

Operator Contract

Operator contracts are owned by oracle node operators, which run alongside offchain oracle nodes.

Request

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

- The oracle address.
- The job ID, so the oracle knows which tasks to perform.
- The callback function, which the oracle sends the response to.

To learn about how to find oracles to suit your needs, see [Find Existing Jobs](#).

Operator contracts are responsible for handling onchain requests made through the LINK token, by implementing `onTokenTransfer` as a `LinkTokenReceiver`. Upon execution of this function, the operator contract emits an `OracleRequest` event containing information about the request. This event is crucial, as it is monitored by the offchain oracle node which acts upon it.

Fulfillment

For fulfillment, the operator 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.

Offchain oracle node

The offchain oracle node is responsible for listening for events emitted by its corresponding onchain 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 operator contract, using the `fulfillOracleRequest` function.

For more information on how to become a node operator, learn how to run a Chainlink node.

Consumer UML

Below is a UML diagram describing the contract structure of `ATestnetConsumer`, a

deployed example contract implementing ChainlinkClient.

```
<ClickToZoom src="/files/8ac3fc1-69a048b-ConsumerUML.svg" />
```

```
# off-chain-reporting.mdx:
```

```
---
section: global
date: Last Modified
title: "Offchain Reporting"
whatsnext: { "Using Data Feeds": "/data-feeds/price-feeds/" }
metadata:
  image: "/files/fb73165-cl.png"
---
```

```
import { Aside } from "@components"
```

Offchain Reporting (OCR) is a significant step towards increasing the decentralization and scalability of Chainlink networks. See the OCR Protocol Paper for a technical deep dive.

For Offchain Reporting aggregators, all nodes communicate using a peer to peer network. During the communication process, a lightweight consensus algorithm runs where each node reports its data observation and signs it. A single aggregate transaction is then transmitted, which saves a significant amount of gas.

The report contained in the aggregate transaction is signed by a quorum of oracles and contains all oracles' observations. By validating the report onchain and checking the quorum's signatures onchain, we preserve the trustlessness properties of Chainlink oracle networks.

What is OCR?

```
<Aside type="note" title="A simple analogy">
  Imagine ordering 10 items from an online store. Each item is packaged
  separately and posted separately, meaning
  postage and packaging costs must be applied to each one, and the carrier has
  to transport 10 different boxes.
  <br />
  OCR, on the other hand, packages all of these items into a single box and
  posts that. This saves postage and packaging
  fees and all effort the carrier associates with transporting 9 fewer boxes.
</Aside>
```

The OCR protocol allows nodes to aggregate their observations into a single report offchain 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 onchain.

Submitting only one transaction per round achieves the following benefits:

- 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 onchain.

How does OCR work?

Protocol execution happens mostly offchain over a peer to peer network between Chainlink nodes. The nodes regularly elect a new leader node that 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 attempt to transmit the final report to the aggregator contract according to a randomized schedule. The aggregator verifies that a quorum of nodes signed the report and exposes the median value to consumers as an answer with a block timestamp and a round ID.

All nodes watch the blockchain for the final report to remove any single point of failure during transmission. If the designated node fails to get their transmission confirmed within a determined period, a round-robin protocol kicks in so other nodes can also transmit the final report until one of them is confirmed.

```
# architecture.mdx:
```

```
---
section: ccip
date: Last Modified
title: "CCIP Architecture"
whatsnext:
  {
    "CCIP manual execution": "/ccip/concepts/manual-execution",
    "Learn CCIP best practices": "/ccip/best-practices",
    "Find the list of supported networks, lanes, and rate limits on the CCIP
Supported Networks page": "/ccip/supported-networks",
  }
---
```

```
import { Aside, ClickToZoom } from "@components"
```

```
<Aside type="note" title="Prerequisites">
  Read the CCIP Introduction and Concepts to understand all the concepts
discussed on this
  page.
</Aside>
```

High-level architecture

Below is a diagram displaying the basic architecture of CCIP. Routers are smart contracts that provide a simple and consistent interface for users. Users can interact with routers to:

- Call smart contract functions on a different blockchain.
- Transfer tokens to a smart contract or Externally Owned Account (EOA) on a different blockchain.
- Send arbitrary messages and tokens within the same transaction. Use this functionality to transfer tokens and instructions on what to do with those tokens to a smart contract on a different blockchain.

```
<Aside type="caution" title="Transferring tokens">
  This section uses the term "transferring tokens" even though the tokens are
not technically transferred. Instead, they
  are locked or burned on the source blockchain and then unlocked or minted on
the destination blockchain. Read the
```

Token Pools section to understand the various mechanisms that are used to transfer value across blockchain.

[Chainlink CCIP Basic Architecture](/images/ccip/ccip-diagram-04v04.webp)

CCIP terminology

CCIP enables a sender on a source blockchain to send a message to a receiver on a destination blockchain.

Term	Description
Sender	A smart contract or an EOA.
Source Blockchain	The blockchain the sender interacts with CCIP from.
Message	Arbitrary data and/or tokens.
Receiver	A smart contract or an EOA. Note: An EOA cannot receive arbitrary data. It can only receive tokens.
Destination Blockchain	The blockchain the receiver resides on.

Detailed architecture

The figure below outlines the different components involved in a cross-chain transaction:

- Cross-Chain dApps are user-specific. A smart contract or an EOA (Externally Owned Account) interacts with the CCIP Router to send arbitrary data and/or transfer tokens cross-chain.
- The contracts in dark blue are the CCIP interface (Router). To use CCIP, users need only to understand how to interact with the router; they don't need to understand the whole CCIP architecture. Note: The CCIP interface is static and remains consistent over time to provide reliability and stability to the users.
- The contracts in light blue are internal to the CCIP protocol and subject to change.

[Chainlink CCIP Detailed Architecture](/images/ccip/detailed-architecture.png)

Onchain components

Router

The Router is the primary contract CCIP users interface with. This contract is responsible for initiating cross-chain interactions. One router contract exists per blockchain. When transferring tokens, callers have to approve tokens for the router contract.

The router contract routes the instruction to the destination-specific OnRamp.

When a message is received on the destination chain, the router is the contract that delivers tokens to the user's account or the message to the receiver's smart contract.

Commit Store

The Committing DON interacts with the CommitStore contract on the destination

blockchain to store the Merkle root of the finalized messages on the source blockchain. This Merkle root must be blessed by the Risk Management Network before the Executing DON can execute them on the destination blockchain. The CommitStore ensures the message is blessed by the Risk Management Network and only one CommitStore exists per lane.

OnRamp

One OnRamp contract per lane exists. This contract performs the following tasks:

- Checks destination-blockchain-specific validity such as validating account address syntax
- Verifies the message size limit and gas limits
- Keeps track of sequence numbers to preserve the sequence of messages for the receiver
- Manages billing
- Interacts with the TokenPool if the message includes a token transfer.
- Emits an event monitored by the committing DON

OffRamp

One OffRamp contract per lane exists. This contract performs the following tasks:

- Ensures the message is authentic by verifying the proof provided by the Executing DON against a committed and blessed Merkle root
- Makes sure transactions are executed only once
- After validation, the OffRamp contract transmits any received message to the Router contract. If the CCIP transaction includes token transfers, the OffRamp contract calls the TokenPool to transfer the correct assets to the receiver.

Token pools

Each token is associated with its own token pool, an abstraction layer over ERC-20 tokens designed to facilitate token-related operations for OnRamping and OffRamping. Token pools provide rate limiting, a security feature enabling token issuers to set a maximum rate at which their token can be transferred per lane. Token pools are configured to lock or burn tokens on the source blockchain and unlock or mint tokens on the destination blockchain. This setup results in four primary mechanisms:

- Burn and Mint: Tokens are burned on the source blockchain, and an equivalent amount of tokens are minted on the destination blockchain.
- Lock and Mint: Tokens are locked on their issuing blockchain, and fully collateralized "wrapped" tokens are minted on the destination blockchain. These wrapped tokens can be transferred across non-issuing blockchains using the Burn and Mint mechanism.
- Burn and Unlock: Tokens are burned on the source blockchain, and an equivalent amount of tokens are released on the destination blockchain. This mechanism is the inverse of the Lock and Mint mechanism. It applies when you send tokens to their issuing source blockchain.
- Lock and Unlock: Tokens are locked on the source blockchain, and an equivalent amount of tokens are released on the destination blockchain.

The mechanism for handling tokens varies depending on the characteristics of each token. Below are several examples to illustrate this:

- LINK Token is minted on a single blockchain (Ethereum mainnet) and has a fixed total supply. Consequently, CCIP cannot natively mint it on another blockchain. For LINK, the token pool is configured to lock tokens on Ethereum mainnet (the issuing blockchain) and mint them on the destination blockchain. Conversely, when transferring from a non-issuing blockchain to Ethereum mainnet, the LINK token pool is set to burn the tokens on the source (non-issuing) blockchain and unlock them on Ethereum Mainnet (issuing). For example, transferring 10 LINK

from Ethereum mainnet to Base mainnet involves the LINK token pool locking 10 LINK on Ethereum mainnet and minting 10 LINK on Base mainnet. Conversely, transferring 10 LINK from Base mainnet to Ethereum mainnet involves the LINK token pool burning 10 LINK on Base mainnet and unlocking 10 LINK on Ethereum mainnet.

- Wrapped native Assets (e.g., WETH) use a Lock and Unlock mechanism. For instance, when transferring 10 WETH from Ethereum mainnet to Optimism mainnet, the WETH token pool will lock 10 WETH on Ethereum mainnet and unlock 10 WETH on Optimism mainnet. Conversely, transferring from Optimism mainnet back to Ethereum mainnet involves the WETH token pool locking 10 WETH on Optimism mainnet and unlocking 10 WETH on the Ethereum mainnet.
- Stablecoins (e.g., USDC) can be minted natively on multiple blockchains. Their respective token pools employ a Burn and Mint mechanism, burning the token on the source blockchain and then minting it natively on the destination blockchain.
- Tokens with a Proof Of Reserve (PoR) with a PoR feed on a specific blockchain present a challenge for the Burn and Mint mechanism when applied across other blockchains due to conflicts with the PoR feed. For such tokens, the Lock and Mint approach is preferred.

```
{/ prettier-ignore /}
```

```
<Aside type="note" title="Transferring Native Gas Tokens">
```

To transfer **ETH**, from one blockchain to another, follow these steps:

1. Wrap ETH into WETH: Users must first interact with the WETH contract or use a DEX to convert their ETH into Wrapped Ether (WETH).

1. Send WETH via CCIP: Next, users call the CCIP Router to send WETH to the receiver account on the desired destination blockchain.

1. Unwrap WETH back into ETH: Finally, on the destination blockchain, users call the WETH contract or use a DEX to convert the WETH into ETH.

Note: You can use the `EthSenderContract.sol` contract as a reference to transfer ETH across blockchains using CCIP.

```
</Aside>
```

```
<section id="onboarding-new-tokens">
```

```
  <Aside type="note" title="Onboarding New Tokens">
```

Supported tokens for each lane are listed on the CCIP supported network pages. Expanding

support for additional tokens is an ongoing process to help ensure the highest level of cross-chain security. Token

pool rate limits are configured per-token for each lane. These rate limits are set together with token issuers where applicable. Rate limits are also selected based on various factors,

such as the unique risk characteristics of the token and the intended use cases. The onboarding process for adding

new tokens also helps ensure that dApps and users interact with the correct token version, reducing risks within the cross-chain ecosystem.

```
  </Aside>
```

```
</section>
```

Risk Management Network contract

The Risk Management contract maintains the list of Risk Management node addresses that are allowed to bless or curse. The contract also holds the quorum logic for blessing a committed Merkle Root and cursing CCIP on a destination blockchain. Read the Risk Management Network Concepts section to learn more.

Offchain components

Committing DON

The Committing DON has several jobs where each job monitors cross-chain transactions between a given source blockchain and destination blockchain:

- Each job monitors events from a given OnRamp contract on the source blockchain.
- The job waits for finality, which depends on the source blockchain.
- The job bundles transactions and creates a Merkle root. This Merkle root is signed by a quorum of oracles nodes part of the Committing DON.
- Finally, the job writes the Merkle root to the CommitStore contract on the given destination blockchain.

Executing DON

Like the Committing DON, the Executing DON has several jobs where each executes cross-chain transactions between a source blockchain and a destination blockchain:

- Each job monitors events from a given OnRamp contract on the source blockchain.
- The job checks whether the transaction is part of the relayed Merkle root in the CommitStore contract.
- The job waits for the Risk Management Network to bless the message.
- Finally, the job creates a valid Merkle proof, which is verified by the OffRamp contract against the Merkle root in the CommitStore contract. After these check pass, the job calls the OffRamp contract to complete the CCIP transactions on the destination blockchain.

Separating commitment and execution permits the Risk Management Network to have enough time to check the commitment of messages before executing them. The delay between commitment and execution also permits additional checks such as abnormal reorg depth, potential simulation, and slashing.

Saving a commitment is compact and has a fixed gas cost, whereas executing user callbacks can be highly gas intensive. Separating commitment and execution permits execution by end users in various cases, such as retrying failed executions.

Risk Management Network

The Risk Management Network is a set of independent nodes that monitor the Merkle roots committed by the Committing DON into the Commit Store.

Each node compares the committed Merkle roots with the transactions received by the OnRamp contract. After the verification succeeds, it calls the Risk Management contract to "bless" the committed Merkle root. When there are enough blessing votes, the root becomes available for execution. In case of anomalies, each Risk Management node calls the Risk Management contract to "curse" the system. If the cursed quorum is reached, the Risk Management contract is paused to prevent any CCIP transaction from being executed.

Read the Risk Management Network Concepts section to learn more.

CCIP rate limits

Chainlink CCIP token transfers benefit from rate limits for additional security. A rate limit has a maximum capacity and a refill rate, which is the speed at which the maximum capacity is restored after a token transfer has consumed some or all of the available capacity.

<ClickToZoom src="/images/ccip/ccip-refill-rate-v03.webp" alt="Chainlink CCIP Detailed Architecture" />

You can find the complete list of lanes and their rate limits on the CCIP Supported Networks page.

The rate limits are enforced at both the source and destination blockchains for maximum security. If these rate limits are reached, descriptive errors with detailed information are generated and returned to the sender. This enables CCIP users to gracefully handle these errors within their dApps to preserve the end-user experience. A comprehensive list of errors and their descriptions is available on the [errors API reference page](#).

Token pool rate limit

For each supported token on each individual lane, the token pool rate limit manages the total number of tokens that can be transferred within a given time. This limit is independent of the USD value of the token.

For example, the maximum capacity of the suUSD token pool on the Ethereum mainnet's Base mainnet lane is 200,000 suUSD. The refill rate is 2 suUSD per second. If 200,000 suUSD are transferred on that lane, the entire capacity is consumed. After that, if a user wants to send 20 suUSD, they must wait at least 10 seconds as the capacity refills. The maximum throughput for this token on the lane is 1200 suUSD every 10 minutes.

Aggregate rate limit

Each lane also has an aggregate rate limit that limits the overall USD value that can be transferred on that lane across all supported tokens. To improve security, the aggregate rate limit for any given lane is always lower than the sum of all individual token pool rate limits for that lane.

Consider an example where a lane has a maximum capacity of 100,000 USD, a refill rate of 167 USD per second, and several token transfers with a total value of 60,000 USD have been executed. In that example, the remaining available capacity is 40,000 USD. If a user intends to transfer tokens equating to 50,000 USD, they must wait at least 60 seconds for capacity to refill the additional 10,000 USD that is required. The maximum throughput in USD value on the lane is 100,000 USD every 10 minutes.

CCIP execution latency

Chainlink CCIP has been purposely designed to take a security-first approach to minimize the risk of block reorgs on the source blockchain. The end-to-end transaction time of a CCIP cross-chain transaction largely depends on the time it takes for the transaction on the source blockchain to reach finality. The time to reach finality varies by blockchain. For example, on Ethereum, it takes about 15 minutes for a block to be finalized. When cross-chain transactions are initiated from a blockchain with faster finality, such as Avalanche, which has a time-to-finality of around one second, the end-to-end transaction time is faster.

If the fee paid on the source blockchain is within an acceptable range of the execution cost on the destination chain, the message will be transmitted as soon as possible after it is blessed by the Risk Management Network. If the cost of execution increases between request and execution time, CCIP incrementally increases the gas price to attempt to reach eventual execution, but this might introduce additional latency.

Execution delays in excess of one hour should be rare as a result of Smart Execution. The Smart Execution time window parameter represents the waiting time before manual execution is enabled. If the DON fails to execute within the duration of Smart Execution time window due to extreme network conditions, you can manually execute the message through the CCIP Explorer. Read the [manual execution conceptual guide](#) to learn more.

CCIP Upgradability

What can be upgraded

Upgradability in the context of CCIP refers to the ability to upgrade CCIP's operational scope, feature set, security, or reliability. These updates are applied via changes to the following items:

- Smart Contract Configuration: Configuration can be updated via dedicated functions exposed by the smart contract. Examples are updates to the rate limits for a specific lane, the list of supported tokens, and the OnRamp contract to call for a given destination blockchain.
- Smart Contract Code: Once deployed, smart contract code cannot be modified or upgraded. The only option is to deploy a new version of the contract (e.g., OnRamp, OffRamp, Token Pool) and redirect the calling contract to the newly deployed contract via a configuration change. One exception is the Router contract, which is the primary interface for CCIP users on both the source and destination blockchains. The contract remains static and consistent over time to ensure reliability and stability for developers who integrate with it.

Implementation process

All onchain configuration changes and upgrades to CCIP must pass through a Role-based Access Control Timelock (RBACTimelock) smart contract.

Any proposal must either (1) be proposed by a dedicated ManyChainMultiSig (MCMS) contract and then be subjected to a review period enforced by the RBACTimelock, during which a quorum of node operators securing CCIP are able to veto the proposal; or (2) be proposed by a dedicated MCMS contract and be explicitly approved by a quorum of independent signers, including multiple node operators securing CCIP, providing an alternative path during time-sensitive circumstances.

Any onchain update that passes the timelock review period without a veto becomes executable by anyone, which can be done by running a timelock-worker to process executable upgrades.

Documentation and source code relating to the CCIP owner contracts can be read on GitHub. The proposer multisig on Ethereum can be found on Etherscan, where configuration details can also be read.

best-practices.mdx:

```
---
section: ccip
date: Last Modified
title: "CCIP Best Practices"
whatsnext: { "Supported networks": "/ccip/supported-networks" }
---
```

```
import { Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<CcipCommon callout="talkToExpert" />
```

```
<Aside type="note" title="Interfaces and Applications">
```

Chainlink CCIP is a messaging protocol. Third parties may build user interfaces or other applications on top of CCIP.

Neither Chainlink Labs nor the Chainlink Foundation owns, controls, endorses, or assumes any responsibility for any

such interfaces or applications. You are solely responsible for your use of such interfaces or applications. Please

visit the Chainlink Foundation Terms of Service for more information.

```
</Aside>
```

Before you deploy your cross-chain dApps to mainnet, make sure that your dApps follow the best practices in this document. You are responsible for thoroughly reviewing your code and applying best practices to ensure that your cross-chain dApps are secure and reliable. If you have a unique use case for CCIP that might involve additional cross-chain risk, contact the Chainlink Labs Team before deploying your application to mainnet.

Verify destination chain

Before calling the router's `ccipSend` function, ensure that your code allows users to send CCIP messages to trusted destination chains.

Example: For an example of how to verify the destination chain, refer to the [Transfer Tokens with Data - Defensive example](#).

Verify source chain

When implementing the `ccipReceive` method in a contract residing on the destination chain, ensure to verify the source chain of the incoming CCIP message. This verification ensures that CCIP messages can only be received from trusted source chains.

Example: For an example of how to verify the source chain, refer to the [Transfer Tokens with Data - Defensive example](#).

Verify sender

When implementing the `ccipReceive` method in a contract residing on the destination chain, it's important to validate the sender of the incoming CCIP message. This check ensures that CCIP messages are received only from trusted sender addresses.

Note: Depending on your use case, this verification might not always be necessary.

Example: For an example of how to verify the sender of the incoming CCIP message, refer to the [Transfer Tokens with Data - Defensive example](#).

Verify router addresses

When you implement the `ccipReceive` method in the contract residing on the destination chain, validate that the `msg.sender` is the correct router address. This verification ensures that only the router contract can call the `ccipReceive` function on the receiver contract and is for developers that want to restrict which accounts are allowed to call `ccipReceive`.

Example: For an example of how to verify the router, refer to the [Transfer Tokens with Data - Defensive example](#).

Setting gasLimit

The `gasLimit` specifies the maximum amount of gas CCIP can consume to execute `ccipReceive()` on the contract located on the destination blockchain. It is the main factor in determining the fee to send a message. Unspent gas is not refunded.

To transfer tokens directly to an EOA as a receiver on the destination blockchain, the `gasLimit` should be set to 0 since there is no `ccipReceive()` implementation to call.

To estimate the accurate gas limit for your destination contract, consider the following options:

- Leveraging Ethereum client RPC by applying `ethestimateGas` on

`receiver.ccipReceive()`. You can find more information on the Ethereum API Documentation and Alchemy documentation.

- Conducting Foundry gas tests.
- Using Hardhat plugin for gas tests.
- Using a blockchain explorer to look up the gas consumption of a particular internal transaction.

Example: For an example of how to estimate the gas limit, refer to the Optimizing Gas Limit Settings in CCIP Messages guide.

Using extraArgs

The purpose of `extraArgs` is to allow compatibility with future CCIP upgrades. To get this benefit, make sure that `extraArgs` is mutable in production deployments. This allows you to build it offchain and pass it in a call to a function or store it in a variable that you can update on-demand.

If `extraArgs` are left empty, a default of 200000 `gasLimit` will be set.

Decoupling CCIP Message Reception and Business Logic

As a best practice, separate the reception of CCIP messages from the core business logic of the contract. Implement 'escape hatches' or fallback mechanisms to gracefully manage situations where the business logic encounters issues. To explore this concept further, refer to the Defensive Example guide.

Evaluate the security and reliability of the networks that you use

Although CCIP has been thoroughly reviewed and audited, inherent risks might still exist based on your use case, the blockchain networks where you deploy your contracts, and the network conditions on those blockchains.

Review and audit your code

Before securing value with contracts that implement CCIP interfaces and routers, ensure that your code is secure and reliable. If you have a unique use case for CCIP that might involve additional cross-chain risk, contact the Chainlink Labs Team before deploying your application to mainnet.

Soak test your dApps

Be aware of the Service Limits and Rate Limits for Supported Networks. Before you provide access to end users or secure value, soak test your cross-chain dApps. Ensure that your dApps can operate within these limits and operate correctly during usage spikes or unfavorable network conditions.

Monitor your dApps

When you build applications that depend on CCIP, include monitoring and safeguards to protect against the negative impact of extreme market events, possible malicious activity on your dApp, potential delays, and outages.

Create your own monitoring alerts based on deviations from normal activity. This will notify you when potential issues occur so you can respond to them.

billing.mdx:

```
---
section: ccip
date: Last Modified
title: "CCIP Billing"
---
```

```
import { Aside } from "@components"
import { Billing, TokenCalculator } from "@features/ccip/components/billing"
```

<Aside type="note" title="Prerequisites">

Read the CCIP Introduction and Concepts to understand all the concepts discussed on this

page.

</Aside>

The CCIP billing model uses the feeToken specified in the message to pay a single fee on the source blockchain. CCIP uses a gas-locked fee payment mechanism, referred to as Smart Execution, to help ensure the reliable execution of cross-chain transactions regardless of destination blockchain gas spikes. For developers, this means you can simply pay on the source blockchain and CCIP will take care of execution on the destination blockchain.

CCIP supports fee payments in LINK and in alternative assets, which currently includes blockchain-native gas tokens and their ERC-20 wrapped versions. The payment model for CCIP is designed to significantly reduce friction for users and quickly scale CCIP to more blockchains by supporting fee payments that originate across a multitude of blockchains over time.

Aside from billing, remember to carefully estimate the gasLimit that you set for your destination contract so CCIP can have enough gas to execute ccipReceive(), if applicable. Any unspent gas from this user-set limit is not refunded.

Billing mechanism

The fee is calculated by the following formula:

$$\text{fee} = \text{blockchain fee} + \text{network fee}$$

Where:

- fee: The total fee for processing a CCIP message. Note: Users can call the getFee function to estimate the fee.
- blockchain fee: This represents an estimation of the gas cost the node operators will pay to deliver the CCIP message to the destination blockchain.
- network fee: Fee paid to CCIP service providers, including node operators running the Decentralized Oracle Network and Risk Management Network.

Blockchain fee

The blockchain fee is calculated by the following formula:

$$\text{blockchain fee} = \text{execution cost} + \text{data availability cost}$$

Execution cost

The execution cost is directly correlated with the estimated gas usage to execute the transaction on the destination blockchain:

$$\text{execution cost} = \text{gas price} \times \text{gas usage} \times \text{gas multiplier}$$

Where:

- gas price: The destination gas price. CCIP maintains a cache of destination gas prices on each source blockchain, denominated in each feeToken.

- gas multiplier: Scaling factor for Smart Execution. This multiplier ensures the reliable execution of transactions regardless of destination blockchain gas spikes.
- gas usage:

gas usage = gas limit + destination gas overhead + destination gas per payload + gas for token transfers

Where:

- gas limit: This specifies the maximum amount of gas CCIP can consume to execute `ccipReceive()` on the receiver contract located on the destination blockchain. Users set the gas limit in the extra argument field of the CCIP message. Note: Remember to carefully estimate the `gasLimit` that you set for your destination contract so CCIP can have enough gas to execute `ccipReceive()`. Any unspent gas from this user-set limit is not refunded.
- destination gas overhead: This is the fixed gas cost incurred on the destination blockchain by CCIP (Committing DON + Executing DON) and Risk Management Network.
- destination gas per payload: This variable gas depends on the length of the data field in the CCIP message. If there is no payload (CCIP only transfers tokens), the value is 0.
- gas for token transfers: This variable gas cost is for transferring tokens onto the destination blockchain. If there are no token transfers, the value is 0.

Data availability cost

This cost is only relevant if the destination blockchain is a L2 layer. Some L2s charge fees for data availability. For instance, optimistic rollups process the transactions offchain then post the transaction data to Ethereum as calldata, which costs additional gas.

Network fee

The fee paid to CCIP service providers, including node operators running the Decentralized Oracle Network and Risk Management Network is calculated as follows:

Token transfers or programmable token transfers

For token transfers or programmable token transfers (token + data), the network fee varies based on the token handling mechanism and the lanes:

- Lock and Unlock: The network fee is percentage-based. For each token, it is calculated using the following expression:

tokenAmount price percentage

Where:

- tokenAmount: The amount of tokens being transferred.
 - price: Initially priced in USD and converted into the feeToken.
 - percentage: The values are provided in the network fee table.
- Lock and Mint, Burn and Mint and Burn and Unlock: The network fee is a static amount. See the network fee table.

<Aside type="note" title="Determine Token Handling Mechanism">

Use the calculator below or consult the CCIP supported networks on the

mainnet or testnet pages to
determine a token's handling mechanism on a given lane.
</Aside>

Messaging (only data)

For messaging (only data): The network fee is a static amount, denominated in USD. See the network fee table.

Network fee table

The table below provides an overview of the network fees charged for different use cases on different lanes. Percentage-based fees are calculated on the value transferred in a message. USD-denominated fees are applied per message.

<Billing />

<br id="network-token-calculator" />

You can use the calculator below to learn the network fees for a specific token. Select the environment (mainnet/testnet), the token, the source blockchain, and the destination blockchain to get the network fee:

<TokenCalculator client:idle />

getting-started.mdx:

```
---
section: ccip
date: Last Modified
title: "Getting Started"
whatsnext:
  {
    "Learn how to transfer tokens": "/ccip/tutorials/cross-chain-tokens",
    "Learn how to transfer tokens and send data in a single CCIP transaction":
"/ccip/tutorials/programmable-token-transfers",
    "Transfer Tokens Between EOAs": "/ccip/tutorials/cross-chain-tokens-from-
eoa",
    "Learn how to send arbitrary data over CCIP": "/ccip/tutorials/send-
arbitrary-data",
    "See the list of supported networks": "/ccip/supported-networks",
  }
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

<CcipCommon callout="talkToExpert" />

A simple use case for Chainlink CCIP is sending data between smart contracts on different blockchains. This guide shows you how to deploy a CCIP sender contract and a CCIP receiver contract to two different blockchains and send data from the sender contract to the receiver contract. You pay the CCIP fees using LINK.

Fees can also be paid in alternative assets, which currently include the native gas tokens of the source blockchain and their ERC20 wrapped version. For example, you can pay ETH or WETH when you send transactions to the CCIP router on Ethereum and AVAX or WAVAX when you send transactions to the CCIP router on Avalanche.

Before you begin

- If you are new to smart contract development, learn how to Deploy Your First Smart Contract so you are familiar with the tools that are necessary for this guide:
 - The Solidity programming language
 - The MetaMask wallet
 - The Remix development environment
- Acquire testnet funds. This guide requires testnet AVAX and LINK on Avalanche Fuji. It also requires testnet ETH on Ethereum Sepolia. If you need to use different networks, you can find more faucets on the LINK Token Contracts page.
 - Go to faucets.chain.link to get your testnet tokens.
- Learn how to Fund your contract with LINK.

Deploy the sender contract

Deploy the Sender.sol contract on Avalanche Fuji. To see a detailed explanation of this contract, read the Code Explanation section.

1. Open the Sender.sol contract in Remix.

```
<CodeSample src="samples/CCIP/Sender.sol" showButtonOnly={true} />
```

1. Compile the contract.

1. Deploy the sender contract on Avalanche Fuji:

1. Open MetaMask and select the Avalanche Fuji network.
1. In Remix under the Deploy & Run Transactions tab, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Avalanche Fuji.
1. Under the Deploy section, fill in the router address and the LINK token contract addresses for your specific blockchain. You can find both of these addresses on the Supported Networks page. The LINK token contract address is also listed on the LINK Token Contracts page. For Avalanche Fuji, the router address is `<CopyText text="0xF694E193200268f9a4868e4Aa017A0118C9a8177" code/>` and the LINK address is `<CopyText text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>`.

```
<ClickToZoom
  src="/images/ccip/tutorials/deploy-sender-avalanche-fuji.webp"
  alt="Chainlink CCIP deploy sender Avalanche Fuji"
/>
```

1. Click the transact button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Avalanche Fuji.

1. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy your contract address.

```
<ClickToZoom
  src="/images/ccip/tutorials/deployed-sender-avalanche-fuji.webp"
  alt="Chainlink CCIP Deployed sender Avalanche Fuji"
  style="max-width: 70%;"
/>
```

1. Open MetaMask and send `<CopyText text="0.1" code/>` LINK to the contract address that you copied. Your contract will pay CCIP fees in LINK.

```
<ClickToZoom
  src="/images/ccip/tutorials/fund-deployed-sender-avalanche-fuji.webp"
  alt="Chainlink CCIP fund deployed sender Avalanche Fuji"
/>
```

Deploy the receiver contract

Deploy the receiver contract on Ethereum Sepolia. You will use this contract to receive data from the sender that you deployed on Avalanche Fuji. To see a detailed explanation of this contract, read the Code Explanation section.

1. Open the Receiver.sol contract in Remix.

```
<CodeSample src="samples/CCIP/Receiver.sol" showButtonOnly={true} />
```

1. Compile the contract.

1. Deploy the receiver contract on Ethereum Sepolia:

1. Open MetaMask and select the Ethereum Sepolia network.

1. In Remix under the Deploy & Run Transactions tab, make sure the Environment is still set to Injected Provider - MetaMask.

1. Under the Deploy section, fill in the router address field. For Ethereum Sepolia, the Router address is `<CopyText text="0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59" code/>`. You can find the addresses for each network on the Supported Networks page.

```
<ClickToZoom
  src="/images/ccip/tutorials/deploy-receiver-sepolia.webp"
  alt="Chainlink CCIP Deploy receiver Sepolia"
/>
```

1. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Ethereum Sepolia.

1. After you confirm the transaction, the contract address appears as the second item in the Deployed Contracts list. Copy this contract address.

```
<ClickToZoom
  src="/images/ccip/tutorials/deployed-receiver-sepolia.webp"
  alt="Chainlink CCIP deployed receiver Sepolia"
  style="max-width: 70%;"
/>
```

You now have one sender contract on Avalanche Fuji and one receiver contract on Ethereum Sepolia. You sent 0.1 LINK to the sender contract to pay the CCIP fees. Next, send data from the sender contract to the receiver contract.

Send data

Send a Hello World! string from your contract on Avalanche Fuji to the contract you deployed on Ethereum Sepolia:

1. Open MetaMask and select the Avalanche Fuji network.
1. In Remix under the Deploy & Run Transactions tab, expand the first contract in the Deployed Contracts section.
1. Expand the sendMessage function and fill in the following arguments:

Argument	Description
Value (Ethereum Sepolia)	

destinationChainSelector	CCIP Chain identifier of the target blockchain. You can find each network's chain selector on the supported networks page

```
<CopyText text="16015286601757825753" code/>
```


receiver	The destination smart contract address
Your deployed contract address	
text	Any string
<CopyText text="Hello World!" code/>	

```
<ClickToZoom
  src="/images/ccip/tutorials/fuji-sendmessage.webp"
  alt="Chainlink CCIP Sepolia send message"
  style="max-width: 70%;"/>
/>
```

1. Click the transact button to run the function. MetaMask prompts you to confirm the transaction.

```
<Aside type="note">
  During gas price spikes, your transaction might fail, requiring more than
0.1 LINK to proceed. If your
  transaction fails, fund your contract with more LINK tokens and try again.
</Aside>
```

1. After the transaction is successful, note the transaction hash. Here is an example of a successful transaction on Avalanche Fuji.

After the transaction is finalized on the source chain, it will take a few minutes for CCIP to deliver the data to Ethereum Sepolia and call the `ccipReceive` function on your receiver contract. You can use the CCIP explorer to see the status of your CCIP transaction and then read data stored by your receiver contract.

1. Open the CCIP explorer and use the transaction hash that you copied to search for your cross-chain transaction. The explorer provides several details about your request.

```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-tx-details.webp"
  alt="Chainlink CCIP Explorer transaction details"/>
/>
```

1. When the status of the transaction is marked with a "Success" status, the CCIP transaction and the destination transaction are complete.

```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-tx-details-success.webp"
  alt="Chainlink CCIP Explorer transaction details success"/>
/>
```

Read data

Read data stored by the receiver contract on Ethereum Sepolia:

1. Open MetaMask and select the Ethereum Sepolia network.
1. In Remix under the Deploy & Run Transactions tab, expand the receiver contract deployed on Ethereum Sepolia.
1. Click the `getLastReceivedMessageDetails` function button to read the stored data. In this example, it is "Hello World!".

```
<ClickToZoom
  src="/images/ccip/tutorials/sepolia-getmessagedetails.webp"
  alt="Chainlink CCIP Sepolia message details"
  style="max-width: 70%;"/>
/>
```

Congratulations! You just sent your first cross-chain data using CCIP. Next, examine the example code to learn how this contract works.

Examine the example code

Sender code

The smart contract in this tutorial is designed to interact with CCIP to send data. The contract code includes comments to clarify the various functions, events, and underlying logic. However, this section explains the key elements. You can see the full contract code below.

```
<CodeSample src="samples/CCIP/Sender.sol" />
```

Initializing the contract

When deploying the contract, you define the router address and the LINK contract address of the blockchain where you choose to deploy the contract.

The router address provides functions that are required for this example:

- The `getFee` function to estimate the CCIP fees.
- The `ccipSend` function to send CCIP messages.

Sending data

The `sendMessage` function completes several operations:

1. Construct a CCIP-compatible message using the `EVM2AnyMessage` struct:

- The receiver address is encoded in bytes format to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through `abi.encode`.
- The data is encoded from a string text to bytes using `abi.encode`.
- The `tokenAmounts` is an array. Each element comprises a struct that contains the token address and amount. In this example, the array is empty because no tokens are sent.
- The `extraArgs` specify the `gasLimit` for relaying the CCIP message to the recipient contract on the destination blockchain. In this example, the `gasLimit` is set to 200000.
- The `feeToken` designates the token address used for CCIP fees. Here, `address(linkToken)` signifies payment in LINK.

1. Compute the fees by invoking the router's `getFee` function.

1. Ensure that your contract balance in LINK is enough to cover the fees.

1. Grant the router contract permission to deduct the fees from the contract's LINK balance.

1. Dispatch the CCIP message to the destination chain by executing the router's `ccipSend` function.

```
<Aside type="caution" title="Sender contract best practices">
```

This example is simplified for learning purposes. For production code, use the following best practices:

- Do not hardcode `extraArgs`: To simplify the example, `extraArgs` are hardcoded in the contract. The recommendation is to make sure `extraArgs` is mutable. For example, you can build `extraArgs` offchain and pass it in your functions call or store it in a storage variable that you can update on demand. Thus, you can make sure `extraArgs` remains backward compatible for future CCIP upgrades.
- Validate the destination chain.

```
</Aside>
```

Receiver code

The smart contract in this tutorial is designed to interact with CCIP to receive data. The contract code includes comments to clarify the various functions, events, and underlying logic. However, this section explains the key elements. You can see the full contract code below.

<CodeSample src="samples/CCIP/Receiver.sol" />

Initializing the contract

When you deploy the contract, you define the router address. The receiver contract inherits from the CCIPReceiver.sol contract, which uses the router address.

Receiving data

On the destination blockchain:

1. The CCIP Router invokes the ccipReceive function. Note: This function is protected by the onlyRouter modifier, which ensures that only the router can call the receiver contract.
1. The ccipReceive function calls an internal function ccipReceive function. The receiver contract implements this function.
1. This ccipReceive function expects an Any2EVMMessage struct that contains the following values:

- The CCIP messageId.
- The sourceChainSelector.
- The sender address in bytes format. The sender is a contract deployed on an EVM-compatible blockchain, so the address is decoded from bytes to an Ethereum address using the ABI specification.
- The data is also in bytes format. A string is expected, so the data is decoded from bytes to a string using the ABI specification.

<Aside type="caution" title="Recommendations Receiver contract">

The example was simplified for learning purposes. For production code, use the following best practices:

- Validate the source chain.
- Depending on your use case, analyze whether you should validate the sender address.

Note that the receiver contract in this example inherits from the base contract CCIPReceiver.sol, which uses the onlyRouter modifier to ensure that only the router can call the ccipReceive function.

</Aside>

index.mdx:

```
---
section: ccip
date: Last Modified
title: "Chainlink CCIP"
isIndex: true
whatsnext:
  {
    "Complete the Getting Started guide to learn the basics": "/ccip/getting-
started",
    "See the list of supported networks": "/ccip/supported-networks",
    "Learn how to transfer tokens": "/ccip/tutorials/cross-chain-tokens",
    "Learn CCIP core concepts": "/ccip/concepts",
  }
```

```
import { ClickToZoom, Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"

<CcipCommon callout="talkToExpert" />
```

Blockchain interoperability protocols are important for the Web3 ecosystem and traditional systems that need to interact with different blockchains. These protocols are the foundation for building blockchain abstraction layers, allowing traditional backends and dApps to interact with any blockchain network through a single middleware solution. Without a blockchain interoperability protocol, Web2 systems and dApps would need to build separate in-house implementations for each cross-chain interaction that they want to use, which is a time-consuming, resource-intensive, and complex process.

Blockchain interoperability protocols provide the following capabilities:

- You can transfer assets and information across multiple blockchains.
- Application developers can leverage the strengths and benefits of different chains.
- Collaboration between developers from diverse blockchain ecosystems enables the building of cross-chain applications to serve more users and provide additional features or products for them.

The Chainlink Cross-Chain Interoperability Protocol (CCIP) provides these capabilities and enables a variety of use cases.

What is Chainlink CCIP?

Chainlink CCIP is a blockchain interoperability protocol that enables developers to build secure applications that can transfer tokens, messages (data), or both tokens and messages across chains.

Given the inherent risks of cross-chain interoperability, CCIP features defense-in-depth security and is powered by Chainlink's industry-standard oracle networks which have a proven track record of securing tens of billions of dollars and enabling over \$14 trillion in onchain transaction value.

CCIP provides several key security benefits:

- Multiple independent nodes run by independent key holders.
- Three decentralized networks all executing and verifying every cross-chain transaction.
- Separation of responsibilities, with distinct sets of node operators, and with no nodes shared between the transactional DONs and the Risk Management Network.
- Increased decentralization with two separate code bases across two different implementations, written in two different languages to create a previously unseen diversity of software clients in the cross-chain world.
- Novel risk management system with level-5 security that can be rapidly adapted to any new risks or attacks that appear in cross-chain messaging.

```
<ClickToZoom src="/images/ccip/ccip-diagram-04v04.webp" alt="Chainlink CCIP Architecture" />
```

To understand how Chainlink CCIP works, refer to the concepts, architecture, and best practices. If you are new to using Chainlink CCIP, read these guides before you deploy any contracts that use CCIP.

Chainlink CCIP core capabilities

Chainlink CCIP supports three main capabilities:

- Arbitrary Messaging: is the ability to send arbitrary data (encoded as bytes)

to a receiving smart contract on a different blockchain. The developer is free to encode any data they wish to send. Typically, developers use arbitrary messaging to trigger an informed action on the receiving smart contract, such as rebalancing an index, minting a specific NFT, or calling an arbitrary function with the sent data as custom parameters. Developers can encode multiple instructions in a single message, enabling them to orchestrate complex, multi-step, multi-chain tasks.

- Token Transfer: You can transfer tokens to a smart contract or directly to an Externally Owned Account (EOA) on a different blockchain.

- Programmable Token Transfer: is the ability to simultaneously transfer tokens and arbitrary data (encoded as bytes) within a single transaction. This mechanism allows users to transfer tokens and send instructions on what to do with those tokens. For example, a user could transfer tokens to a lending protocol with instructions to leverage those tokens as collateral for a loan, borrowing another asset to be sent back to the user.

It is recommended that token issuers use CCIP's native Token Transfer capabilities to enable the transfer of their tokens across blockchains. CCIP Token Transfers offer rigorously audited Token Pool contracts and provide configurable rate-limiting functionalities for an enhanced developer experience and risk management. In addition, Token Transfers also improve the composability of tokens with other dApps and token bridges that are integrated with the standardized CCIP interface.

Receiving account types

With CCIP, you send transactions with data, tokens, or both data and tokens. The receiver of a CCIP transaction is either a smart contract or an externally owned account (EOA). Smart contracts can receive both data and tokens, while EOAs can only receive tokens:

CCIP capability supported	What is sent	Type of receiving account
Arbitrary Messaging	Data	Smart contracts only. EOAs on EVM blockchains cannot receive messages.
Token Transfer	Tokens	Smart contracts and EOAs
Programmable Token Transfer	Data and tokens	Smart contracts only. If you send data and transfer tokens to an EOA, only tokens will be transferred.

Common use cases

Chainlink CCIP enables a variety of use cases:

- Cross-chain lending: Chainlink CCIP enables users to lend and borrow a wide range of crypto assets across multiple DeFi platforms running on independent chains.
- Low-cost transaction computation: Chainlink CCIP can help offload the computation of transaction data on cost-optimized chains.
- Optimizing cross-chain yield: Users can leverage Chainlink CCIP to move collateral to new DeFi protocols to maximize yield across chains.
- Creating new kinds of dApps: Chainlink CCIP enables users to take advantage of network effects on certain chains while harnessing compute and storage capabilities of other chains.

Read [What Are Cross-Chain Smart Contracts](#) to learn about cross-chain smart contracts and examples of use cases they enable.

Supported networks

See the [Supported Networks](#) page for a list of supported networks, tokens, and

contract addresses.

To learn about tokens, token pools, and the token onboarding process, see the [CCIP Architecture](#) page.

release-notes.mdx:

```
---
section: ccip
date: Last Modified
title: "Chainlink CCIP Release Notes"
---
```

```
import { Aside } from "@components"
```

Metis - 2024-08-08

Chainlink CCIP is publicly available on Metis mainnet and testnet.

Blast - 2024-07-09

Chainlink CCIP is publicly available on Blast mainnet and Sepolia testnet.

Mode - 2024-06-19

Chainlink CCIP is publicly available on Mode mainnet and Sepolia testnet.

Gnosis mainnet - 2024-06-05

Chainlink CCIP is publicly available on Gnosis mainnet.

Celo - 2024-05-29

Chainlink CCIP is publicly available on Celo mainnet and Alfajores testnet.

Polygon Amoy - 2024-05-08

Chainlink CCIP is publicly available on Polygon Amoy.

See the [CCIP testnet configuration](#) page for more information.

Chainlink CCIP is GA - 2024-04-24

Chainlink CCIP is now Generally Available (GA) on mainnet and testnet.

To support your development and implementation needs, we encourage you to reach out to our team of experts for guidance and support. For expert advice, visit the [Chainlink CCIP Contact form](#).

Additionally, the Chainlink CCIP local simulator is available to enhance your development workflow with CCIP. This tool allows you to simulate Chainlink CCIP functionality locally within your Hardhat and Foundry projects. The simulator is designed so you can test your contracts locally and transition smoothly to test networks without any modifications.

Support for WETH transfers and gas limit mainnet increase - 2024-04-11

WETH and support of Lock and Unlock mechanism

Chainlink's CCIP now supports WETH (Wrapped Ether) transfers through the Lock and Unlock token mechanism. This feature allows CCIP to securely lock tokens on the source blockchain and subsequently release an equivalent amount of tokens on the destination blockchain, facilitating seamless cross-chain transfers of WETH.

The introduction of this mechanism enables WETH transfers across several key lanes:

- Ethereum Mainnet to/from Arbitrum Mainnet.
- Ethereum Mainnet to/from Optimism Mainnet.
- Arbitrum Mainnet to/from Optimism Mainnet.

For a specific lane configuration, go to the [Mainnet Supported Networks](#) page. For more detailed information on the Lock and Unlock mechanism and its applications, read the [Token Pools](#) section.

Gas limit increase on mainnet

The maximum gasLimit that you can set for CCIP messages on mainnet has been increased to 3,000,000 gas units. The change has been documented in the [Service Limits](#) page.

v1.0.0 deprecated on mainnet - 2024-04-01

CCIP v1.0.0 is no longer supported on mainnet. You must use the new router addresses listed in the [CCIP v1.2.0 configuration](#) page.

Wemix and Kroma - 2024-03-11

Chainlink CCIP is publicly available on Wemix and Kroma for both mainnet and testnet.

v1.0.0 deprecated on testnet - 2024-02-07

CCIP v1.0.0 is no longer supported on testnet. You must use the new router addresses listed in the [CCIP v1.2.0 configuration](#) page.

v1.2.0 release on mainnet - 2024-01-15

<Aside type="caution" title="Deprecation Notice for CCIP v1.0.0 on Mainnet">

CCIP v1.0.0 has been deprecated on mainnet. You must use the new router addresses listed in this page before March

31st, 2024. Please note that there is no change to the router interface. The CCIP v1.0.0 mainnet routers will

continue to function in parallel until March 31st, 2024, but we highly recommend switching to the v1.2.0 routers

as soon as possible. If you currently use CCIP v1.0.0, use the @chainlink/contracts-ccip npm package version

0.7.6. To migrate to v1.2.0, use version 1.2.1 of the npm package or later. Please refer to the

release notes for a comprehensive overview of the enhancements and new features in v1.2.0.

</Aside>

- There is no change to the router interface, but you must use the new router addresses listed in the [CCIP v1.2.0 configuration](#) page.

- Added support for USDC transfers. USDC transfers are currently supported on the following lanes:

- From Ethereum Mainnet to Avalanche Mainnet.
- From Ethereum Mainnet to Arbitrum Mainnet.
- From Ethereum Mainnet to Base Mainnet.
- From Ethereum Mainnet to Optimism Mainnet.
- From Optimism Mainnet to Base Mainnet.
- From Optimism Mainnet to Ethereum Mainnet.
- From Avalanche Mainnet to Ethereum Mainnet.

- From Base Mainnet to Arbitrum Mainnet.
- From Base Mainnet to Optimism Mainnet.
- From Base Mainnet to Ethereum Mainnet.

Refer to the Supported Networks to get a specific lane's token addresses and rate limits.

- We've simplified the message sequencing process in our CCIP message handling by removing the strict sequencing flag from the extraArgs field in CCIP messages.

- The gas limit and maximum message data length for CCIP messages have been adjusted on mainnets. These changes are detailed in the Service Limits documentation.

- To interact with CCIP v1.2.0 , use the @chainlink/contract-ccip npm package.

Arbitrum Sepolia - 2023-12-15

Chainlink CCIP is publicly available on Arbitrum Sepolia.

See the CCIP testnet configuration page for more information.

v1.2.0 release on testnet - 2023-12-08

<Aside type="caution" title="Deprecation Notice for CCIP v1.0.0 on Testnet">

CCIP v1.0.0 has been deprecated on testnet. You must use the new router addresses listed in the CCIP v1.2.0

configuration page before January 31st, 2024. Please note that there is no change to the router interface. The CCIP v1.0.0 testnet routers will continue to function in parallel until

January 31st, 2024, but we highly recommend switching to the v1.2.0 routers as soon as possible. If you currently

use CCIP v1.0.0, use the @chainlink/contracts-ccip npm package version

0.7.6. To migrate to v1.2.0, use version 1.2.1 of

the npm package or later.

</Aside>

- There is no change to the router interface, but you must use the new router addresses listed in the CCIP v1.2.0 configuration page.

- Added support for USDC transfers. USDC transfers are currently supported on the following lanes:

- From Optimism Goerli to Avalanche Fuji.
- From Optimism Goerli to Base Goerli.
- From Avalanche Fuji to Optimism Goerli.
- From Avalanche Fuji to Base Goerli.
- From Base Goerli to Optimism Goerli.
- From Base Goerli to Avalanche Fuji.

Refer to the Supported Networks to get a specific lane's token addresses and rate limits.

- We've simplified the message sequencing process in our CCIP message handling by removing the strict sequencing flag from the extraArgs field in CCIP messages.

- The gas limit and maximum message data length for CCIP messages have been adjusted on testnets. These changes are detailed in the Service Limits documentation.

- To interact with CCIP v1.2.0 , use the @chainlink/contract-ccip npm package.

Arbitrum Goerli - 2023-11-17

Arbitrum Goerli is no longer supported.

Arbitrum Sepolia support will be added at a later date.

New mainnets - 2023-09-27

Added BNB and Base mainnets:

- BNB mainnet.
- Base mainnet.

See the supported networks page for more information.

New mainnet - 2023-09-21

Added Arbitrum mainnet:

- Arbitrum mainnet.

See the supported networks page for more information.

New testnets - 2023-08-25

Added BNB and Base testnets:

- BNB testnet.
- Base Goerli.

See the supported networks page for more information.

Testnet GA release - 2023-07-20

Chainlink CCIP is publicly available on the following testnet chains:

- Ethereum Sepolia.
- Optimism Goerli.
- Avalanche Fuji.
- Arbitrum Goerli.
- Polygon Mumbai.

See the supported networks page for more information.

Mainnet Early Access - 2023-07-17

```
<Aside type="note" title="Mainnet Access">  
  Apply for Mainnet Early Access to get started.  
</Aside>
```

Chainlink CCIP is available on the following networks:

- Ethereum mainnet.
- Optimism mainnet.
- Avalanche mainnet.
- Polygon mainnet.

See the supported networks page for more information.

service-limits.mdx:

section: ccip

date: Last Modified
title: "CCIP Service Limits"

```
import { Aside } from "@components"  
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

Mainnet

<Aside type="note" title="Increase the gas limit">
If you consistently need more than 3M for your use case, please reach out to your Chainlink Labs point of contact or contact us via this contact form.
</Aside>

Item	Description
Limit	

Maximum message data length	data payload sent within the CCIP message
30 kilobytes	
Message Gas Limit	User specified gas limit
3,000,000	
Maximum number of tokens a user can transfer in a single transaction	Maximum number of distinct tokens
	1
Smart Execution time window	Maximum duration for the execution of a CCIP message
8 hours	

Testnet

Item	Description
Limit	

Maximum message data length	data payload sent within the CCIP message
30 kilobytes	
Message Gas Limit	User specified gas limit
3,000,000	
Maximum number of tokens a user can transfer in a single transaction	Maximum number of distinct tokens
	1
Smart Execution timeframe	Maximum duration for the execution of a CCIP message
8 hours	

service-responsibility.mdx:

section: ccip
date: Last Modified
title: "Chainlink CCIP Service Responsibility"

```
import { Aside } from "@components"
```

<Aside type="note" title="Terms Of Service">
Chainlink does not hold or transfer any assets. Please review the [Chainlink Terms of Service](https://chain.link/terms) which provides important information and disclosures.
By using Chainlink CCIP, you expressly acknowledge and agree to accept these terms.

</Aside>

The Chainlink Cross-Chain Interoperability Protocol (CCIP) is a secure, reliable, and easy-to-use interoperability protocol for building cross-chain applications and services. The use of CCIP involves application developers, blockchain development teams, and Chainlink node operators, among others. These participants share responsibility for ensuring that operation and performance match expectations. Please note that CCIP support of a particular blockchain, application, or token does not constitute endorsement of such blockchain, application, or token.

Application Developer Responsibilities

Application developers are responsible for the correctness, security, and reliability of their application. This includes:

- Code and application audits: Developers are responsible for auditing their code and applications before deploying to production. Developers must determine the quality of any audits and ensure that they meet the requirements for their application.
- CCIP upgrades and best practices: Developers are responsible for following CCIP documentation regarding implementing CCIP upgrades and best practices for integrating CCIP in their applications.
- Code dependencies and imports: Developers are responsible for ensuring the quality, reliability, and security of any dependencies or imported packages that they use with Chainlink CCIP, as well as reviewing and auditing these dependencies and packages.
- Code quality and testing: Developers are responsible for ensuring that their application code, onchain and offchain, meets the quality expectations and has undergone rigorous testing.
- Application monitoring and alerting: Developers must monitor their applications, inform their users of any abnormal activity, and take appropriate action to restore normal operations.
- Blockchain risk assessment: Developers are responsible for the risk assessment of any blockchain network where they choose to deploy their application on or decide to interoperate with, when using Chainlink CCIP. This includes reviewing the time-to-finality formally documented by a blockchain's development team, which is used to inform how long CCIP waits for finality when outbound transactions are initiated from that chain.
- Token risk assessment: Developers are responsible for the risk assessment of any tokens they choose to support or list in their application and expose to their users.
- Risk communication: Developers must clearly articulate and communicate identified risks to their users.
- Manual execution: Developers must monitor their CCIP transactions and take action when transactions require manual execution. For example, informing their users and directing them to the appropriate page on the CCIP Explorer.

Blockchain Development Team Responsibilities

Blockchain development teams are responsible for the correctness, security, and reliability of their blockchain software. This includes:

- Block finality: Blockchain development teams must ensure that blocks with a commitment level of finalized are actually final. The properties of the finality mechanism, including underlying assumptions and conditions under which finality violations could occur, must be clearly documented and communicated to application developers in the blockchain ecosystem. The documented time-to-finality informs how long CCIP waits for finality for outbound transactions from that chain; however, an additional buffer may be added.
- Governance model: Blockchain development teams are responsible for setting up a clear and effective governance model and communicating its participants and processes clearly to its stakeholders and application developers.
- Fixes and upgrades: Blockchain development teams must communicate availability

of fixes immediately and announce planned upgrades as much in advance as possible so blockchain validators and application developers can prepare themselves accordingly.

- Incident management: Blockchain development teams are responsible for clearly articulating and communicating any security, reliability and availability incidents to their community. This includes root cause analysis, post-mortem details and a clear plan of action to recover and prevent from happening in the future.
- Blockchain liveness: Blockchain development teams must take appropriate action to ensure their blockchain maintains a high degree of liveness and aligns with set expectations towards their community members and applications developers.

Chainlink Node Operator Responsibilities

High-quality Chainlink node operators participate in the decentralized oracle networks (DONs) that power CCIP and the Risk Management Network using a configuration specified in the Chainlink software. As participants in these deployments, Node Operators are responsible for the following components of Chainlink CCIP and the Risk Management Network:

- Node operations: Chainlink node operators must ensure the proper configuration, maintenance, and monitoring of their nodes participating in the Chainlink CCIP and Risk Management Network DONs.
- Transaction execution: Chainlink node operators must ensure that transactions execute onchain in a timely manner and that they apply gas bumping when necessary.
- Blockchain client: Chainlink node operators are responsible for selecting and properly employing blockchain clients, including latest fixes and upgrades, to connect to supported blockchain networks.
- Consensus participation: Chainlink node operators must maintain continuous uptime and active participation in OCR consensus.
- Infrastructure security: Chainlink node operators must follow infrastructure security best practices. These include access control, configuration management, key management, software version & patch management, and (where applicable) physical security of the underlying hardware.
- Software version: Chainlink node operators are responsible for ensuring that Chainlink node deployments are running the latest software versions.
- Responsiveness: Chainlink node operators must respond to important communication from Chainlink Labs or from other node operators in a timely manner.

```
# test-tokens.mdx:
```

```
---
section: ccip
date: Last Modified
title: "Acquire Test Tokens"
---
```

```
import { ClickToZoom, Aside } from "@components"
import { MintTokenButton } from "@features/ccip/components/MintTokenButton"
```

As a best practice, always test your applications thoroughly on testnet before going live on mainnet. When testing token transfers, you must have enough tokens and ensure the token pools have enough funds. Public faucets sometimes limit how many tokens a user can create and token pools might not have enough liquidity. To resolve these issues, CCIP supports two test tokens that you can mint permissionlessly so you don't run out of tokens while testing different scenarios.

Tokens

Two ERC20 test tokens are currently available on each testnet. You can find the

token addresses for each testnet on the Supported Networks page.

Name	Decimals	Description
Type		

<div style="white-space: nowrap;">CCIP-BnM</div> <div style="text-align: center;">18</div> <div style="white-space: nowrap;">Burn & Mint</div> These tokens are minted on each testnet. When transferring these tokens between testnet blockchains, CCIP burns the tokens on the source chain and mint them on the destination chain.		
<div style="white-space: nowrap;">CCIP-LnM</div> <div style="text-align: center;">18</div> <div style="white-space: nowrap;">Lock & Mint</div> These tokens are only minted on Ethereum Sepolia. On other testnet blockchains, the token representation is a wrapped/synthetic asset called clCCIP-LnM. When transferring these tokens from Ethereum Sepolia to another testnet, CCIP locks the CCIP-LnM tokens on the source chain and mint the wrapped representation clCCIP-LnM on the destination chain. Between non-Ethereum Sepolia chains, CCIP burns and mints the wrapped representation clCCIP-LnM.		

Mint Test Tokens

You can mint both of these tokens using the following function call on the token contract. This function acts like a faucet. Each call mints 10¹⁸ units of a token to the specified address.

- For CCIP-BnM, you can call drip on all testnet blockchains.
- For CCIP-LnM, you can call drip only on Ethereum Sepolia.

```
solidity
function drip(address to) external {
    mint(to, 1e18);
}
```

Mint tokens in the documentation

You can use this interface to connect your MetaMask wallet, select a testnet, and mint tokens to your wallet address. Alternatively, you can call these same functions in the block explorer (Read the Mint tokens in a block explorer section).

```
<MintTokenButton client:visible />
```

Mint tokens in a block explorer

Follow these steps to learn how to mint these tokens. The steps explain how to mint CCIP-BnM on Ethereum Sepolia:

- Go to the Supported Networks page.
- Go to Ethereum Sepolia section. You will find a list of active lanes where the source chain is Ethereum Sepolia. You will find the list of supported tokens you can transfer for each lane. You should find CCIP-BnM in the list for each testnet.
- Click on the token address to display it on the block explorer (CCIP-BnM on

Ethereum Sepolia Etherscan in this case).

```
<ClickToZoom src="/images/ccip/test-tokens/token-etherscan.jpg" alt="Chainlink  
CCIP Test token etherscan" />
```

- Click the Contract tab and then on Write Contract to see the list of transactions. Notice Connect to Web3.

```
<ClickToZoom  
  src="/images/ccip/test-tokens/token-etherscan-connect-to-web3.jpg"  
  alt="Chainlink CCIP Test token etherscan connect to web3."  
/>
```

- Click Connect to Web3 to connect your MetaMask wallet to the block explorer.
- Once connected, you can call the drip function.

```
<ClickToZoom  
  src="/images/ccip/test-tokens/token-etherscan-connected-to-web3.jpg"  
  alt="Chainlink CCIP Test token etherscan connected to web3."  
/>
```

- Fill in the text field with your EOA, then click Write.
- MetaMask will open and asks you to confirm the transaction.
- After the transaction is confirmed, click View your transaction to view your transaction.

```
<ClickToZoom  
  src="/images/ccip/test-tokens/token-etherscan-view-transaction.jpg"  
  alt="Chainlink CCIP Test token etherscan view your transaction."  
/>
```

- You should see a successful transaction confirming that 1 CCIP-BnM was sent to your EOA.

```
<ClickToZoom  
  src="/images/ccip/test-tokens/token-etherscan-token-minted.jpg"  
  alt="Chainlink CCIP Test token etherscan minted."  
/>
```

- Follow this MetaMask guide to import CCIP-BnM in your wallet.

```
# ccip-receiver.mdx:
```

```
---  
section: ccip  
date: Last Modified  
title: "CCIPReceiver API Reference"  
---
```

```
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<CcipCommon callout="importCCIPPackage" />
```

CCIP receiver contracts inherit from CCIPReceiver.

```
solidity  
import {CCIPReceiver} from  
"@chainlink/contracts-ccip/src/v0.8/ccip/applications/CCIPReceiver.sol";  
...
```

```
constructor(address router) is CCIPReceiver(router) {  
}
```

Functions

constructor

```
solidity
constructor(address router) internal
```

supportsInterface

```
solidity
function supportsInterface(bytes4 interfaceId) public pure returns (bool)
```

IERC165 supports an interfaceId

Parameters

Name	Type	Description
interfaceId	bytes4	The interfaceId to check

Return Values

Name	Type	Description
[0]	bool	true if the interfaceId is supported

ccipReceive

```
solidity
function ccipReceive(struct Client.Any2EVMMessage message) external override
onlyRouter
```

Only the Router can call this function to deliver a message.
If this reverts, any token transfers also revert. The message
will move to a FAILED state and become available for manual execution.

Parameters

Name	Type	Description
message	struct Client.Any2EVMMessage	CCIP Message

\ccipReceive

```
solidity
function ccipReceive(struct Client.Any2EVMMessage message) internal virtual
```

Override this function in your implementation.

Parameters

Name	Type	Description
message	struct Client.Any2EVMMessage	CCIP Message

```
| message | struct Client.Any2EVMMessage | Any2EVMMessage |
```

getRouter

solidity

```
function getRouter() public view returns (address)
```

This function returns the current Router address.

Return Values

Name	Type	Description
[0]	address	irouter address

InvalidRouter

solidity

```
error InvalidRouter(address router)
```

onlyRouter

solidity

```
modifier onlyRouter()
```

Only calls from the set router are accepted.

client.mdx:

section: ccip

date: Last Modified

title: "Client Library API Reference"

```
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<CcipCommon callout="importCCIPPackage" />
```

CCIP senders and receivers use the CCIP Client Library to build CCIP messages.

solidity

```
import { Client } from
```

```
"@chainlink/contracts-ccip/src/v0.8/ccip/libraries/Client.sol";
```

Types and Constants

EVMTokenAmount

Use this solidity struct to specify the token address and amount.

solidity

```
struct EVMTokenAmount {
```

```
    address token;
```

```
    uint256 amount;
```

```
}
```

Name	Type	Description
------	------	-------------

	-----		-----		-----	
	token		address		token address on the local chain.	
	amount		uint256		Amount of tokens.	

Any2EVMMessage

CCIP receivers use this solidity struct to parse the received CCIP message.

```

solidity
struct Any2EVMMessage {
    bytes32 messageId;
    uint64 sourceChainSelector;
    bytes sender;
    bytes data;
    struct Client.EVMTokenAmount[] destTokenAmounts;
}

```

Name	Type	Description
-----	-----	-----

messageId	bytes32	CCIP messageId, generated on the source chain.
sourceChainSelector	uint64	Source chain selector.
sender	bytes	Sender address.
data	bytes	Payload sent within the CCIP message.
destTokenAmounts	Client.EVMTokenAmount[]	Tokens and their amounts in their destination chain representation.

EVM2AnyMessage

CCIP senders use this solidity struct to build the CCIP message.

```

solidity
struct EVM2AnyMessage {
    bytes receiver;
    bytes data;
    struct Client.EVMTokenAmount[] tokenAmounts;
    address feeToken;
    bytes extraArgs;
}

```

Name	Type	Description
-----	-----	-----

receiver	bytes	Receiver address. Use abi.encode(sender) to encode the address to bytes.
data	bytes	Payload sent within the CCIP message.
tokenAmounts	Client.EVMTokenAmount[]	Tokens and their amounts in the source chain representation.
feeToken	address	Address of feeToken. Set address(0) to pay in native gas tokens such as ETH on Ethereum or POL on Polygon.

| extraArgs | bytes | Users fill in the EVMExtraArgsV1 struct then encode it to bytes using the \argsToBytes function |

EVMEExtraArgsV1TAG

```
solidity
bytes4 EVMEExtraArgsV1TAG
```

EVMEExtraArgsV1

```
solidity
struct EVMEExtraArgsV1 {
    uint256 gasLimit;
}
```

Name	Type	Description
-----	-----	

gasLimit	uint256	specifies the maximum amount of gas CCIP can consume to execute ccipReceive() on the contract located on the destination blockchain. Read Setting gasLimit for more details.
----------	---------	--

Functions

\argsToBytes

```
solidity
function argsToBytes(struct Client.EVMEExtraArgsV1 extraArgs) internal pure
returns (bytes bts)
```

It is used to convert the arguments to bytes.

Parameters

Name	Type	Description
extraArgs	Client.EVMEExtraArgsV1	Extra arguments.

Return Values

Name	Type	Description
bts	bytes	Encoded extra arguments in bytes.

errors.mdx:

```
---
section: ccip
date: Last Modified
title: "Errors API Reference"
---
```

```
import CcipCommon from "@features/ccip/CcipCommon.astro"
import { CCIPSendError } from "@features/ccip/components/api-reference"
import { Aside, CopyText } from "@components"
```

<CcipCommon callout="importCCIPPackage" />

When invoking the `ccipSend` function, it is possible to encounter various errors. These might be thrown either by the CCIP router or by one of the downstream contracts called by the CCIP router. Below is a compiled list of potential errors you might encounter. Referencing this list will enable you to capture and handle these exceptions gracefully.

Router

```
<CCIPSendError type="router" />
```

Onramp

```
<CCIPSendError type="onramp" />
```

RateLimiter

```
<CCIPSendError type="rate-limiter" />
```

ERC20

```
<CCIPSendError type="erc20" />
```

PriceRegistry

```
<CCIPSendError type="price-registry" />
```

```
# i-router-client.mdx:
```

```
---
section: ccip
date: Last Modified
title: "IRouterClient API Reference"
---
```

```
import { Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<CcipCommon callout="importCCIPPackage" />
```

To send messages through CCIP, users must interact with the `IRouterClient` interface.

After you import `IRouterClient.sol`, you can initialize a router client instance:

```
solidity
import {IRouterClient} from
"@chainlink/contracts-ccip/src/v0.8/ccip/interfaces/IRouterClient.sol";
...
IRouterClient router;
constructor(address router) {
    router = IRouterClient(router);
}
```

Errors

UnsupportedDestinationChain

```
solidity
error UnsupportedDestinationChain(uint64 destChainSelector)
```

InsufficientFeeTokenAmount

```
solidity
error InsufficientFeeTokenAmount()
```

InvalidMsgValue

```
solidity
error InvalidMsgValue()
```

Functions

isChainSupported

```
solidity
function isChainSupported(uint64 chainSelector) external view returns (bool
supported)
```

Checks if the given chain ID is supported for sending/receiving.

Parameters

Name	Type	Description
-----	-----	-----
chainSelector	uint64	The chain to check.

Return Values

Name	Type	Description
-----	-----	-----
supported	bool	is true if supported or false if not.

getSupportedTokens

```
solidity
function getSupportedTokens(uint64 chainSelector) external view returns
(address[] tokens)
```

Gets a list of all supported tokens which can be sent or received to or from a given chain ID.

Parameters

Name	Type	Description
-----	-----	-----
chainSelector	uint64	The chainSelector.

Return Values

Name	Type	Description
-----	-----	-----
tokens	address[]	The addresses of all supported tokens.

getFee

```
solidity
function getFee(uint64 destinationChainSelector, struct Client.EVM2AnyMessage
message) external view returns (uint256 fee)
```

returns 0 fees on invalid message.

Parameters

Name	Type	Description
destinationChainSelector	uint64	The destination chainSelector
message	struct Client.EVM2AnyMessage	The cross-chain CCIP message, including data and/or tokens

Return Values

Name	Type	Description
fee	uint256	returns guaranteed execution fee for the specified message delivery to the destination chain

ccipSend

solidity

```
function ccipSend(uint64 destinationChainSelector, struct Client.EVM2AnyMessage message) external payable returns (bytes32)
```

Request a message to be sent to the destination chain.

<Aside type="caution">

If the msg.value exceeds the required fee from getFee, the overpayment is accepted with no refund.

</Aside>

Parameters

Name	Type	Description
destinationChainSelector	uint64	The destination chain ID
message	struct Client.EVM2AnyMessage	The cross-chain CCIP message, including data and/or tokens

Return Values

Name	Type	Description
[0]	bytes32	messageId The message ID

index.mdx:

section: ccip

date: Last Modified

title: "CCIP Conceptual Overview"

isIndex: true

whatsnext: { "CCIP Architecture": "/ccip/architecture", "Learn CCIP best

```
practices": "/ccip/best-practices" }  
---
```

Before you explore how Chainlink CCIP works in the architecture guide, it is best to understand the core concepts.

Prerequisites

Before you learn about Chainlink CCIP, you should understand these fundamental blockchain concepts:

- Accounts: you should understand the differences between Externally-owned accounts (EOA) and Contract accounts.
- Smart Contracts.
- Decentralized Applications (dApps).
- ERC-20 Token Standard.
- Merkle Trees.

Cross-Chain dApps

Cross-Chain Decentralized Applications (Cross-Chain dApps) are decentralized applications designed to interact with multiple blockchain networks using cross-chain interoperability protocols. Each application leverages the unique strengths of its underlying blockchain network, providing a unified and seamless user experience. Cross-Chain dApps enable transactions, communication, and data sharing between blockchains, increasing functionality and cooperation between multiple blockchain ecosystems.

Interoperability

Interoperability is the ability to exchange information between different systems or networks, even if they are incompatible. Shared concepts on different networks ensure that each party understands and trusts the exchanged information. It also considers the concept of finality to establish trust in the exchanged information by validating its accuracy and integrity.

The web3 ecosystem has become multi-chain, with the rise of layer-1 blockchains and layer-2 scaling solutions, where each network has its own approach to scalability, security, and trust. However, blockchains are isolated networks that operate independently and cannot communicate natively with traditional systems to make external API calls or interact with other blockchains. This limitation is known as the oracle problem. This limitation creates the need for blockchain interoperability protocols, which enable connectivity between different blockchain networks.

CCIP uses interoperability to help web3 developers overcome blockchain restraints, enable specialization of web3 applications, and allow dApps to leverage the liquidity and benefits of multiple blockchain ecosystems.

To learn more, read about blockchain interoperability and cross-chain smart contracts.

Finality

Finality is the assurance that past transactions included onchain are extremely difficult or impossible to revert. If the parameters for finality are properly set, the likelihood of reversibility is extremely low. For CCIP, source chain finality is the main factor that determines the end-to-end elapsed time for CCIP to send a message from one chain to another.

Finality varies across different networks. Some networks offer instant finality and others require multiple confirmations. These time differences are set to ensure the security of CCIP and its users. Finality is crucial for token transfers because funds are locked and not reorganized once they are released.

onto the destination chain. In this scenario, finality ensures that funds on the destination chain are available only after they have been successfully committed on the source chain.

Lane

A Chainlink CCIP lane is a distinct pathway between a source and a destination blockchain. Lanes are unidirectional. For instance, Ethereum Mainnet => Polygon Mainnet and Polygon Mainnet => Ethereum Mainnet are two different lanes.

Decentralized Oracle Network (DON)

Chainlink Decentralized Oracle Networks, or DONs, run Chainlink OCR2. The protocol runs in rounds during which an observed data value might be agreed upon. The output of this process results in a report which is attested to by a quorum of participants. The report is then transmitted onchain by one of the participants. No single participant is responsible for transmitting on every round, and all of them will attempt to do so in a round-robin fashion until a transmission has taken place. In the context of CCIP, a lane contains two OCR DON committees that monitor transactions between a source and destination blockchain: the Committing DON and Executing DON. Read the Architecture page to learn more.

Risk Management Network

The Risk Management Network is built using offchain and onchain components:

- Offchain: Several Risk Management nodes continually monitor all supported chains against abnormal activities
- onchain: One Risk Management contract per supported CCIP chain

Offchain Risk Management node

The Risk Management Network is a secondary validation service parallel to the primary CCIP system. It doesn't run the same codebase as the DON to mitigate against security vulnerabilities that might affect the DON's codebase.

The Risk Management Network has two main modes of operation:

- Blessing: Each Risk Management node monitors all Merkle roots of messages committed on each destination chain. The Committing DON commits these Merkle roots. (More information on Merkle roots can be found on the architecture page). The Risk Management node independently reconstructs the Merkle tree by fetching all messages on the source chain. Then, it checks for a match between the Merkle root committed by the Committing DON and the root of the reconstructed Merkle tree. If both Merkle roots match, the Risk Management node blesses the root to the Risk Management contract on the destination chain. The Risk Management contract tracks the votes. When a quorum is met, the Risk Management contract dubs the Merkle root blessed.

- Cursing: If a Risk Management node detects an anomaly, the Risk Management node will curse the CCIP system. After a quorum of votes has been met, the Risk Management contract dubs the CCIP system cursed. CCIP will automatically pause on that chain and wait until the contract owner assesses the situation before potentially lifting the curse. There are two cases where Risk Management nodes pause CCIP:

- Finality violation: A deep reorganization which violates the safety parameters set by the Risk Management configuration occurs on a CCIP chain.

- Execution safety violation: A message is executed on the destination chain without any matching transaction being on the source chain. Double executions fall into this category since the executing DON can only execute a message once.

Onchain Risk Management contract

There is one Risk Management contract for each supported destination chain. The Risk Management contract maintains a group of nodes authorized to participate in the Risk Management blessing/cursing.

Each Risk Management node has five components:

- An address for voting to curse
- An address for voting to bless
- An address for withdrawing a vote to curse
- A curse weight
- A blessing weight

The contract also maintains two thresholds to determine the quorum for blessing and cursing. There are two different voting logics depending on the mode:

- Blessing voting procedure: every time a Risk Management node blesses a Merkle root, the Risk Management contract adds the blessing weight for that node. If the sum of the weights of votes to bless exceeds the blessing threshold, the Risk Management contract considers the contract blessed.
- Cursing voting procedure: a Risk Management node that sends a vote to curse assigns the vote a random 32-byte ID. The node may have multiple active votes to curse at any time. However, if there is at least one active cursing vote, the Risk Management contract considers the node to have voted to curse. The Risk Management contract adds the cursing weight for that node. If the sum of the weights of votes to curse exceeds the curse threshold, the Risk Management contract considers the contract cursed.

If the Risk Management contract is cursed, then the owner of the original contract must resolve any underlying issues the original contract might have. If the owner is satisfied that these issues have been resolved, they can revoke the cursing on behalf of Risk Management nodes.

```
# manual-execution.mdx:
```

```
---
section: ccip
date: Last Modified
title: "CCIP Manual Execution"
whatsnext:
  { "Manual execution guide": "/ccip/tutorials/manual-execution", "Learn CCIP
best practices": "/ccip/best-practices" }
---
```

```
import { Aside, ClickToZoom, CopyText } from "@components"
```

```
<Aside type="note" title="Prerequisites">
  Read the CCIP Concepts and Architecture pages to understand all the concepts
  discussed on this page.
</Aside>
```

In general, messages are successfully delivered and processed via CCIP as described in the Architecture page. CCIP has a built-in gas bumping mechanism called Smart Execution to ensure that messages are reliably delivered to the destination blockchain. Read the CCIP Execution latency section to learn more about Smart Execution. However, some exceptional conditions might require users to manually execute the transaction on the destination blockchain:

- The receiver contract on the destination blockchain reverted due to an unhandled exception such as a logical error.
- The receiver contract on the destination blockchain reverted due to the gas limit being insufficient to execute the triggered function (Note: The gas limit value is set in the extraArgs param of the message).

- The message could not be executed on the destination chain within CCIP's Smart Execution time window, which is currently set to 8 hours. This could happen, for example, during extreme network congestion and resulting gas spikes.

The flowchart below displays the process of a cross-chain transaction, detailing the steps involved in the manual execution:

<ClickToZoom src="/images/ccip/manual-execution.png" alt="Chainlink CCIP manual execution flowchart" />

CCIP execution

1. A sender contract or EOA initiates a CCIP message on the source blockchain.
1. CCIP Committing DON awaits finality on the source blockchain.
1. Post finality, the CCIP Committing DON assembles a batch of eligible transactions, computes a Merkle root, and records it to the CommitStore contract on the destination blockchain.
1. After successful verification, the Risk Management Network blesses the committed Merkle root.
1. After the committed Merkle root is blessed, the CCIP Executing DON proceeds with the execution on the destination blockchain:
 - During the Smart Execution time window, the Executing DON assesses and, if necessary, adjusts the gas price (a process known as gas bumping) to facilitate successful transaction execution on the destination blockchain as part of the Smart Execution process.
 - If the Smart Execution time window has passed and the CCIP message has still not been executed, then Manual Execution is enabled. See the Manual execution section for more details.
1. The execution on the destination blockchain works as follows:

1. If the message involves token transfers, the tokens are first transferred to the receiver.

- If the receiver is an EOA, then this transaction is considered complete with no further processing.
- If the receiver is a smart contract, the ccipReceive function is invoked after the token transfer. The ccipReceive function processes the CCIP message and any user-specified logic in the receiver contract. The execution of the CCIP message is atomic (all or none). If the ccipReceive function successfully executes, then all aspects of the transaction are complete, and there is no revert. If, however, there is an issue in the receiver execution due to insufficient gas limit or unhandled exceptions, the attempted token transfer will also revert. The transaction then becomes eligible for manual execution.

1. If the message does not involve token transfers, only arbitrary messaging, and the receiver execution fails due to gas limits or unhandled exceptions, the transaction becomes eligible for manual execution.

Manual execution

As described above, a CCIP message becomes eligible for manual execution for various reasons. Manual execution means that a user has to manually trigger the execution of the destination transaction after the issue that caused manual execution has been resolved.

When a CCIP message is eligible for manual execution, the CCIP explorer shows the following information:

- Ready for manual execution status
- The possibility to override the gas limit and a Trigger Manual Execution button

<ClickToZoom src="/images/ccip/manual-execution-status.jpg" alt="Chainlink CCIP manual execution status" />

Depending on the situation, you can take one of the following steps:

- Insufficient gas limit: The executor can connect their wallet, override the gas limit parameter, increase the gas limit for this particular execution, and trigger a manual execution. If this new gas limit override is sufficient, the transaction will go through successfully. Note: This gas limit override applies only to this manually executed transaction.
- Unhandled exception (logical error) in the receiver contract: If the receiver contract is upgradeable, developers must correct the logic, re-deploy the logic contract, and then manually execute the same transaction. If the receiver contract is not upgradeable, developers must deploy a new receiver contract, and then users can send a new CCIP message. Non-upgradable contracts will not benefit from manual execution. Note: Always make sure to test your smart contracts thoroughly. As a best practice, implement fallback mechanisms in the CCIP receiver contracts to manage unhandled exceptions gracefully. Read the Defensive example to learn more.
- If the issue was due to extreme gas spikes or network conditions, and CCIP was not able to successfully transfer the message despite gas bumping for the entire duration of the Smart Execution time window (currently 8 hours), then you can try manual execution.

When manual execution is initiated, a Merkle proof is generated for the message to be executed. During execution, the CCIP explorer submits the Merkle proof and the new gas limit (if the initial failure was due to a low gas limit). This Merkle proof is verified by the OffRamp contract against the Merkle root in the CommitStore contract, and that was blessed by the Risk Management Network. This mirrors the automated execution performed by the CCIP Executing DON, with the addition that the execution is resubmitted using the gas limit you provide.

Frequently asked questions

1. Can anyone execute a transaction on the CCIP explorer even if they are not the initiator of the transaction?

Yes, any EOA can manually execute a CCIP message that is eligible for manual execution. However, the executing account must have sufficient native gas tokens (such as ETH on Ethereum or POL on Polygon) to cover the gas costs associated with the delivery of the CCIP message.

1. If a user sends multiple messages and the first message isn't successfully delivered and goes into a manual execution mode, does that mean all subsequent messages from the user will also be stuck?

It depends. If a message goes into manual execution mode due to receiver errors (unhandled exceptions or gas limit issues), subsequent messages don't get automatically blocked, unless they would encounter the same error. However, suppose a message goes into manual execution mode after the Smart Execution time window expires (currently 8 hours). In that case, subsequent messages must wait for the first message to be processed to maintain the default sequence.

1. If the maximum gas limit is 3M (3,000,000) on mainnet, but it turns out that the destination blockchain requires more than that, will an override of > 3M still work?

Yes, but only for this execution. This works because the gas limit for this execution instance isn't passing through the CCIP validation for the gas limit, for which the CCIP executing DON pays the gas. However, if you consistently need more than 3M for your use case, please reach out to us via this contact form.

1. Will Chainlink Labs reimburse us for manual execution fees?

Since most manual execution situations are due to insufficient gas limit or an unhandled exception in the receiver contract, Chainlink Labs does not reimburse these fees. If you are a dApp developer, please ensure you test thoroughly to avoid manual executions to the extent possible.

1. Do I have to manually execute via the CCIP explorer? Are there any other ways to do this?

The CCIP explorer provides the easiest way to execute manually. It handles all the complexity of submitting the Merkle proof needed for successful transaction execution.

1. How do I know if my receiver error is due to a gas limit issue or an unhandled exception?

If you see a ReceiverError with a revert reason as empty (0x), this is likely due to a gas limit issue. You can look at the transaction trace for the destination transaction on a tool such as Tenderly, and you will likely see an out of gas reason mentioned in such cases. Determine the gas limit that would work for your transaction and manually override it. Read the manual execution tutorial to analyze an example of an exception due to a low gas limit.

1. How can I write contracts that avoid manual execution situations in the first place?

- Test thoroughly to ensure logical conditions for all paths are gracefully handled in your receiver contract.

- Set a gas limit that works for complex execution paths, not just the simplest ones. Read the best practices for gas estimation.

- Refer to the Defensive example tutorial for an example of how to design a programmable token transfer that handles errors gracefully.

1. My transaction meets one of the conditions to trigger manual execution, but I do not see this option on the CCIP explorer. How am I supposed to execute this manually?

This should not happen, but in the unlikely scenario that it does, please submit a support ticket as shown below. Include the CCIP Message ID, your preferred contact details, and a detailed description of the issue you faced. This will help us assist you more effectively.

!CCIP manual execution support ticket

index.mdx:

```

section: ccip
date: Last Modified
title: "Example Cross-chain dApps and Tools"
isIndex: true
whatsnext: { "Supported networks": "/ccip/supported-networks", "Learn CCIP best
practices": "/ccip/best-practices" }
---
```

```

import { Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<CcipCommon callout="talkToExpert" />
```

Several example dApps and tools are available to help you learn about use cases for CCIP.

CCIP Starter Kits

The CCIP Starter Kits demonstrate how to transfer tokens and send messages using the HardHat or Foundry frameworks.

- HardHat CCIP Starter Kit
- Foundry CCIP Starter Kit

CCIP Tic Tac Toe

CCIP Tic Tac Toe demonstrates how to build a gaming dApp that operates across multiple blockchain networks.

Cross-chain name service

The Cross-chain Name Service is an educational example of how to create a minimal cross-chain name service using Chainlink CCIP.

DeFi lending

The DeFi Lending examples shows how a cross-chain lending application can work using CCIP.

DeFi liquidation protection

The DeFi liquidation protection example shows how a DeFi dApp can use CCIP to prevent liquidation when lending assets across multiple blockchain networks.

Cross Chain NFT

The Cross Chain NFT example shows you how to mint an NFT on one blockchain from another blockchain.

```
# index.mdx:
```

```

---
```

```

section: ccip
date: Last Modified
title: "CCIP Supported Networks"
isIndex: true
---
```

- Mainnet
- Testnet

To learn about tokens, token pools, and the token onboarding process, see the CCIP Architecture page.

```
# mainnet.mdx:
```

```
---
```

```
section: ccip
date: Last Modified
title: "CCIP Supported Networks Mainnet"
---
```

```
import { Aside } from "@components"
import Main from "@features/ccip/components/supported-networks/Main.astro"
import { Environment, Version } from "@config/data/ccip"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<CcipCommon callout="talkToExpert" />
```

```
<CcipCommon callout="supportedNetworksConcepts" />
```

Configuration

```
<Main environment={Environment.Mainnet} version={Version.V120} />
```

```
# testnet.mdx:
```

```
---
```

```
section: ccip
date: Last Modified
title: "CCIP Supported Networks Testnet"
---
```

```
import { Aside } from "@components"
import Main from "@features/ccip/components/supported-networks/Main.astro"
import { Environment, Version } from "@config/data/ccip"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<CcipCommon callout="talkToExpert" />
```

```
<CcipCommon callout="supportedNetworksConcepts" />
```

Configuration

```
<Aside type="note" title="Testnet tokens">
```

```
  Read the LINK Token Contracts page to learn where to get testnet LINK
  and native gas tokens. You can pay fees for each network in native gas tokens,
  wrapped native gas tokens, or LINK
  tokens.
```

```
</Aside>
```

```
<Main environment={Environment.Testnet} version={Version.V120} />
```

```
# ccipreceive-gaslimit.mdx:
```

```
---
```

```
section: ccip
date: Last Modified
title: "Optimizing Gas Limit Settings in CCIP Messages"
---
```

When constructing a CCIP message, it's crucial to set the gas limit accurately. The gas limit represents the maximum amount of gas consumed to execute the ccipReceive function on the CCIP Receiver, which influences the transaction fees

for sending a CCIP message. Notably, unused gas is not reimbursed, making it essential to estimate the gas limit carefully:

- Setting the gas limit too low will cause the transaction to revert when CCIP calls `ccipReceive` on the CCIP Receiver, which requires a manual re-execution with an increased gas limit. For more details about this scenario, read the Manual Execution guide.
- Conversely, an excessively high gas limit leads to higher fees.

This tutorial shows you how to estimate the gas limit for the `ccipReceive` function using various methods. You will learn how to use a CCIP Receiver where the gas consumption of the `ccipReceive` function varies based on the input data. This example emphasizes the need for testing under diverse conditions. This tutorial includes tasks for the following environments:

1. Local Environment: Using Hardhat and Foundry on a local blockchain provides a swift initial gas estimate. However, different frameworks can yield different results and the local environment will not always be representative of your destination blockchain. Consider these figures to be preliminary estimates. Then, incorporate a buffer and conduct subsequent validations on a testnet.
1. Testnet: You can precisely determine the required gas limit by deploying your CCIP Sender and Receiver on a testnet and transmitting several CCIP messages with the previously estimated gas. Although this approach is more time-intensive, especially if testing across multiple blockchains, it offers enhanced accuracy.
1. Offchain Methods: Estimating gas with an offchain Web3 provider or tools like Tenderly offers the most accurate and swift way to determine the needed gas limit.

These approaches will give you insights into accurately estimating the gas limit for the `ccipReceive` function, ensuring cost-effective CCIP transactions.

Before you begin, open a terminal, clone the smart-contract-examples repository, and navigate to the `smart-contract-examples/ccip/estimate-gas` directory.

```
bash
git clone https://github.com/smartcontractkit/smart-contract-examples.git &&
\
cd smart-contract-examples/ccip/estimate-gas
```

Examine the code

The source code for the CCIP Sender and Receiver is located in the `contracts` directory for Hardhat projects and in the `src` directory for Foundry projects. The code includes detailed comments for clarity and is designed to ensure self-explanatory functionality. This section focuses on the `ccipReceive` function:

```
solidity
function ccipReceive(Client.Any2EVMMessage memory any2EvmMessage) internal
override {
    uint256 iterations = abi.decode(any2EvmMessage.data, (uint256));

    uint256 result = iterations;
    uint256 maxIterations = iterations % 100;
    for (uint256 i = 0; i < maxIterations; i++) {
        result += i;
    }

    emit MessageReceived(
        any2EvmMessage.messageId,
        any2EvmMessage.sourceChainSelector,
        abi.decode(any2EvmMessage.sender, (address)),
        iterations,
```

```

        maxIterations,
        result
    );
}

```

The ccipReceive function operates as follows:

1. Input Processing: The function accepts a Client.Any2EVMMMessage. The first step involves decoding the number of iterations from the message's data using ABI decoding.

1. Logic Execution: It initializes the result variable with the number of iterations. The function calculates maxIterations by taking the modulo of iterations with 100, which sets an upper limit for iteration. This step is a safeguard to ensure that the function does not run out of gas.

1. Iteration: The function executes a loop from 0 to maxIterations, simulating variable computational work based on the input data. This variability directly influences gas consumption.

1. Event Emission: Finally, an event MessageReceived is emitted.

This code shows how gas consumption for the ccipReceive function can fluctuate in response to the input data, highlighting the necessity for thorough testing under different scenarios to determine the correct gasLimit.

Gas estimation in a local environment

To facilitate testing within a local environment, you will use the MockCCIPRouter contract. This contract serves as a mock implementation of the CCIP Router contract, enabling the local testing of CCIP Sender and Receiver contracts.

A notable feature of the MockCCIPRouter contract is its ability to emit a MsgExecuted event:

```

solidity
event MsgExecuted(bool success, bytes retData, uint256 gassed))

```

This event reports the amount of gas consumed by the ccipReceive function.

Foundry

Prerequisites

1. In your terminal, change to the foundry directory:

```

bash
cd foundry

```

1. Ensure Foundry is installed.

1. Check the Foundry version:

```

bash
forge --version

```

The output should be similar to the following:

```

text
forge 0.2.0 (545cd0b 2024-03-14T00:20:00.210934000Z)

```

You need version 0.2.0 or above. Run foundryup to update Foundry if

necessary.

1. Build your project:

```
bash
forge build
```

The output should be similar to:

```
text
[â $] Compiling...
[â ~] Compiling 52 files with 0.8.19
[â `] Solc 0.8.19 finished in 2.55s
Compiler run successful!
```

Estimate gas

Located in the test directory, the `SendReceive.t.sol` test file assesses the gas consumption of the `ccipReceive` function. This file features a test case that sends a CCIP message to the `MockCCIPRouter` contract, which triggers the `MsgExecuted` event. This event provides insights into the gas requirements of the `ccipReceive` function by detailing the amount of gas consumed. The test case explores three scenarios to examine gas usage comprehensively across various operational conditions:

- Baseline gas consumption: This scenario runs 0 iteration to determine the baseline gas consumption, representing the least amount of gas required.
- Average gas consumption: This scenario runs 50 iterations to estimate the gas consumption under average operational conditions.
- Peak gas consumption: This scenario runs 99 iterations to estimate the peak gas consumption, marking the upper limit of gas usage.

To run the test, execute the following command:

```
bash
forge test -vv --isolate
```

Output example:

```
text
[â $] Compiling...
[â ~] Compiling 52 files with 0.8.19
[â f] Solc 0.8.19 finished in 2.72s
Compiler run successful!
```

Ran 3 tests for test/SendReceive.t.sol:SenderReceiverTest

[PASS] testSendReceiveAverage() (gas: 125166)

Logs:

Number of iterations 50 - Gas used: 14740

[PASS] testSendReceiveMax() (gas: 134501)

Logs:

Number of iterations 99 - Gas used: 24099

[PASS] testSendReceiveMin() (gas: 115581)

Logs:

Number of iterations 0 - Gas used: 5190

Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 10.84ms (5.28ms CPU time)

Ran 1 test suite in 188.81ms (10.84ms CPU time): 3 tests passed, 0 failed, 0 skipped (3 total tests)

This table summarizes the gas usage for different iterations:

Scenario	Number of iterations	Gas used
Baseline gas consumption	0	5190
Average gas consumption	50	14740
Peak gas consumption	99	24099

The output demonstrates that gas consumption increases with the number of iterations, peaking when the iteration count reaches 99. In the next section, you will compare these results with those obtained from a local Hardhat environment.

Hardhat

Prerequisites

1. In your terminal, navigate to the hardhat directory:

```
bash
cd ../hardhat
```

1. Install the dependencies:

```
bash
npm install
```

1. Set the password to encrypt your environment variables using the following command:

```
bash
npx env-enc set-pw
```

1. Set the following environment variables to deploy contracts on testnets:

- PRIVATEKEY: The private key for your testnet wallet. If you use MetaMask, follow the instructions to Export a Private Key. Note: Your private key is needed to sign any transactions you make such as making requests.
- ETHEREUMSEPOLIARPCURL: The RPC URL for Ethereum Sepolia testnet. You can sign up for a personal endpoint from Alchemy, Infura, or another node provider service.
- AVALANCHEFUJIRPCURL: The RPC URL for Avalanche Fuji testnet. You can sign up for a personal endpoint from Infura or another node provider service.
- ETHERSCANAPIKEY: An Ethereum explorer API key, used to verify your contract. Follow this guide to get one from Etherscan.

Input these variables using the following command:

```
bash
npx env-enc set
```

1. Compile the contracts:

```
bash
npx hardhat compile
```

The output should be similar to:

```
text
Generating typings for: 31 artifacts in dir: typechain-types for target:
ethers-v6
Successfully generated 114 typings!
Compiled 33 Solidity files successfully (evm target: paris).
```

Estimate gas

Located in the test directory, the Send-Receive.ts test file is designed to evaluate the gas usage of the ccipReceive function. This file employs the same logic as the Foundry test file, featuring three scenarios varying by the number of iterations. The test case transmits a CCIP message to the MockCCIPRouter contract, triggering the MsgExecuted event. This event provides insights into the gas requirements of the ccipReceive function by detailing the amount of gas used.

To run the test, execute the following command:

```
bash
npx hardhat test
```

Example of the output:

```
text
Sender and Receiver
Final Gas Usage Report:
Number of iterations 0 - Gas used: 5168
Number of iterations 50 - Gas used: 14718
Number of iterations 99 - Gas used: 24077
    " should CCIP message from sender to receiver (1716ms)
```

1 passing (2s)

This table summarizes the gas usage across different iterations:

Scenario	Number of iterations	Gas used
Baseline gas consumption	0	5168
Average gas consumption	50	14718
Peak gas consumption	99	24077

The output demonstrates that gas consumption increases with the number of iterations, peaking when the iteration count reaches 99.

Compare the results from Foundry and Hardhat

This table summarizes the gas usage for different iterations from both Foundry and Hardhat:

Scenario	Number of iterations	Gas used (Foundry)	Gas used (Hardhat)
Baseline gas consumption	0	5190	5168
Average gas consumption	50	14740	14718

Gas usage trends across different iterations are consistent between Foundry and Hardhat and increase with the number of iterations, reaching a peak at 99. However, slight variations in gas usage between the two environments at each iteration level demonstrate the importance of extending gas usage estimation beyond local environment testing. To accurately determine the appropriate gas limit, it is recommended to conduct additional validations on the target blockchain. Setting the gas limit with a buffer is advisable to account for differences between local environment estimations and actual gas usage on the target blockchain.

Estimate gas usage on your local environment

Now that you've locally estimated the gas usage of the `ccipReceive` function using the provided projects, you can apply the same approach to your own Foundry or Hardhat project. This section will guide you through estimating gas usage in your Foundry or Hardhat project.

Estimate `ccipReceive` gas usage locally in your Foundry project

To estimate the gas usage of the `ccipReceive` function within your own Foundry project, follow these steps:

1. Create a testing file in the test directory of your project and import the `MockCCIPRouter` contract:

```
solidity
import { MockCCIPRouter } from
"@chainlink/contracts-ccip/src/v0.8/ccip/test/mocks/MockRouter.sol";
```

Note: The `MockCCIPRouter` receives the CCIP message from your CCIP Sender, calls the `ccipReceive` function on your CCIP Receiver, and emits the `MsgExecuted` event with the gas used.

1. Inside the `setUp` function, deploy the `MockCCIPRouter` contract, and use its address to deploy your CCIP Sender and CCIP Receiver contracts. For more details, check this example.

1. In your test cases:

1. Before transmitting any CCIP messages, use `vm.recordLogs()` to start capturing events. For more details, check this example.

1. After sending the CCIP message, use `vm.getRecordedLogs()` to collect the recorded logs. For more details, check this example.

1. Parse the logs to find the `MsgExecuted(bool,bytes,uint256)` event and extract the gas used. For more details, check this example.

Estimate `ccipReceive` gas usage locally in your Hardhat project

To estimate the gas usage of the `ccipReceive` function within your own Hardhat project, follow these steps:

1. Create a Solidity file in the contracts directory of your project and import the `MockCCIPRouter` contract:

```
solidity
import { MockCCIPRouter } from
"@chainlink/contracts-ccip/src/v0.8/ccip/test/mocks/MockRouter.sol";
```

Note: The MockCCIPRouter receives the CCIP message from your CCIP Sender, calls the ccipReceive function on your CCIP Receiver, and emits the MsgExecuted event with the gas used.

1. Create a testing file in your project's test directory.

1. Inside the deployFixture function, deploy the MockCCIPRouter contract and use its address to deploy your CCIP Sender and CCIP Receiver contracts. For more details, check this example.

1. In your test cases:

1. Send the CCIP message to the MockCCIPRouter contract. For more details, check this example.

1. Parse the logs to find the MsgExecuted(bool,bytes,uint256) event and extract the gas used. For more details, check this example.

Gas estimation on a testnet

To accurately validate your local environment's gas usage estimations, follow these steps:

1. Deploy and configure the CCIP Sender contract on the Avalanche Fuji testnet and the CCIP Receiver contract on the Ethereum Sepolia testnet.

1. Send several CCIP messages with the same number of iterations used in your local testing. For this purpose, use the sendCCIPMessage.ts script in the scripts/testing directory. This script includes a 10% buffer over the estimated gas usage to ensure a sufficient gas limit. Refer to the table below for the buffered gas limits for each iteration:

Scenario	Number of iterations	Estimated gas usage
(Hardhat) Buffered gas limit (+10%)		
-----	-----	-----
Baseline gas consumption	0	5168
5685		
Average gas consumption	50	14718
16190		
Peak gas consumption	99	24077
26485		

1. Use Tenderly to monitor and confirm that the transactions execute successfully within the buffered gas limits. Subsequently, compare the actual gas usage of the ccipReceive function on the Ethereum Sepolia testnet against the buffered limits to fine-tune the final gas limit.

This approach ensures that your gas limit settings are validated against real-world conditions on testnets, providing a more accurate and reliable estimation for deploying on live blockchains.

Deploy and configure the contracts

To deploy and configure the CCIP Sender contract on the Avalanche Fuji testnet and the CCIP Receiver contract on the Ethereum Sepolia testnet, follow the steps below. Note: Your account must have some ETH tokens on Ethereum Sepolia and AVAX tokens on Avalanche Fuji.

1. Deploy the CCIP Sender on the Avalanche Fuji testnet:

```
bash
npx hardhat run scripts/deployment/deploySender.ts --network avalancheFuji
```

1. Deploy the CCIP Receiver on the Ethereum Sepolia testnet:

```

bash
npx hardhat run scripts/deployment/deployReceiver.ts --network
ethereumSepolia

```

1. Authorize the Sender to send messages to Ethereum Sepolia:

```

bash
npx hardhat run scripts/configuration/allowlistingForSender.ts --network
avalancheFuji

```

1. Authorize the Receiver to receive messages from the Sender:

```

bash
npx hardhat run scripts/configuration/allowlistingForReceiver.ts --network
ethereumSepolia

```

Upon completion, you will find the CCIP Sender and Receiver contracts deployed and configured on their respective testnets. Contract addresses are available in the scripts/generatedData.json file.

Send CCIP Messages

1. Send three CCIP messages with different numbers of iterations:

```

bash
npx hardhat run scripts/testing/sendCCIPMessages.ts --network avalancheFuji

```

Example output:

```

text
$ npx hardhat run scripts/testing/sendCCIPMessages.ts --network avalancheFuji
Approving 0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846 for
0x32A24e40851E19d1eD2a7E697d1a38228e9388a3. Allowance is
115792089237316195423570985008687907853269984665640564039457584007913129639935.
Signer 0x9d087fC03ae39b088326b67fA3C788236645b717...

115792089237316195423570985008687907853269984665640564039457584007913129639935n

```

```

Number of iterations 0 - Gas limit: 5685 - Message Id:
0xf23b17366d69159ea7d502835c4178a1c1d1d6325edf3d91dca08f2c7a2900f7
Number of iterations 50 - Gas limit: 16190 - Message Id:
0x4b3a97f6ac959f67d769492ab3e0414e87fdd9c143228f9c538b22bb695ca728
Number of iterations 99 - Gas limit: 26485 - Message Id:
0x37d1867518c0f8c54ceb0c5507b46b8d44c6c53864218f448cba0234f8de867a

```

1. Open the CCIP explorer, search each message by its ID, and wait for each message to be successfully transmitted (Status in the explorer: Success).

For the example above, here are the destination transaction hashes:

Message id	Ethereum Sepolia transaction hash
0xf23b17366d69159ea7d502835c4178a1c1d1d6325edf3d91dca08f2c7a2900f7	0xf004eb6dab30b3cfb9d1d631c3f9832410b8d4b3179e65b85730563b67b1e689
0x4b3a97f6ac959f67d769492ab3e0414e87fdd9c143228f9c538b22bb695ca728	0xf004eb6dab30b3cfb9d1d631c3f9832410b8d4b3179e65b85730563b67b1e689
0x37d1867518c0f8c54ceb0c5507b46b8d44c6c53864218f448cba0234f8de867a	

0xf004eb6dab30b3cfb9d1d631c3f9832410b8d4b3179e65b85730563b67b1e689 |

Note that the Ethereum Sepolia transaction hash is the same for all the messages. This is because CCIP batched the messages.

Check the actual gas usage

1. Open Tenderly and search for the destination transaction hash.

1. Search for `callWithExactGasSafeReturnData` with a payload containing your `messageId` (without `0x`). Example for `0xf23b17366d69159ea7d502835c4178a1c1d1d6325edf3d91dca08f2c7a2900f7`.

1. Below the payload with your `messageId`, you will find the call trace from the Router to your Receiver contract. Call trace example.

1. Click on the Debugger tab and you'll get the gas details:

```
text
{
  "gas": {
    "gasleft": 5685
    "gasused": 5031
    "totalgasused": 7994315
  }
}
```

1. Note the `gasleft` is equal to the limit that is set in the `sendCCIPMessages.ts` script: 5685. The `gasused` is the actual gas used by the Receiver contract to process the message.

1. Repeating the same steps for the other two messages, we can summarize the output:

Scenario	Number of iterations	Estimated gas usage
(Hardhat) Buffered gas limit (+10%) Gas used on testnet		
-----	-----	-----
Baseline gas consumption	0	5168
5685	5031	
Average gas consumption	50	14718
16190	14581	
Peak gas consumption	99	24077
26485	23940	

Testing on testnets has confirmed that a gas limit of 26,485 is adequate for the `ccipReceive` function to execute successfully under various conditions. However, it is important to note that gas usage may differ across testnets. Therefore, it is advisable to conduct similar validation efforts on the blockchain where you intend to deploy. Deploying and validating contracts across multiple testnets can be time-consuming. For efficiency, consider using offchain methods to estimate gas usage.

Offchain methods

This section guides you through estimating gas usage using two different offchain methods:

- A Web3 provider using the `ethers.js` `estimateGas` function.
- Tenderly simulation API. The Tenderly simulation API provides a more accurate result (Read this blog post to learn more) but you are limited to the blockchains supported by Tenderly.

These methods provide the most accurate and rapid means to determine the

necessary gas limit for the `ccipReceive` function. You will use the same CCIP Receiver contract deployed on the Ethereum Sepolia testnet in the previous section.

Prerequisites

1. In your terminal, navigate to the `offchain` directory:

```
bash
cd ../offchain
```

1. Modify the `data.json` file to insert the deployed addresses of your Sender and Receiver contracts.

1. Install the dependencies:

```
bash
npm install
```

1. Set the password to encrypt your environment variables:

```
bash
npx env-enc set-pw
```

1. Set up the following environment variables:

- `ETHEREUMSEPOLIARPCURL`: The RPC URL for Ethereum Sepolia testnet. You can sign up for a personal endpoint from Alchemy, Infura, or another node provider service.
- `TENDERLYACCOUNTSLUG`: This is one part of your Tenderly API URL. You can find this value in your Tenderly account.
- `TENDERLYPROJECTSLUG`: This is one part of your Tenderly API URL. You can find this value in your Tenderly account.
- `TENDERLYACCESSKEY`: If you don't already have one, you can generate a new access token.

Input these variables using the following command:

```
bash
npx env-enc set
```

1. Generate Typechain typings for the Receiver contract:

```
bash
npm run generate-types
```

Introduction of the scripts

The scripts are located in the `src` directory. Each script is self-explanatory and includes comprehensive comments to explain its functionality and usage. There are three scripts:

- `estimateGasProvider.ts`: This script uses the `ethestimateGas` Ethereum API to estimate the gas usage of the `ccipReceive` function. It simulates sending three CCIP messages to the Receiver contract with a varying number of iterations and estimates the gas usage using the `ethers.js` `estimateGas` function.
- `estimateGasTenderly.ts`: This script leverages the Tenderly `simulate` API to estimate the gas usage of the `ccipReceive` function. Similar to the previous

script, it simulates sending three CCIP messages to the Receiver contract with different numbers of iterations and estimates the gas usage using the Tenderly simulate API.

- helper.ts: This script contains helper functions used by the other scripts. The two main functions are:

- buildTransactionData: This function constructs a CCIP message for a specified number of iterations and then returns the transaction data.
- estimateIntrinsicGas: Exclusively called by the estimateGasProvider.ts script, this function estimates the intrinsic gas of a transaction. The intrinsic gas represents the minimum amount of gas required before executing a transaction. It is determined by the transaction data and the type of transaction. Since this gas is paid by the initiator of the transaction, we use this function to estimate the intrinsic gas and then deduct it from the total gas used to isolate the gas consumed by the ccipReceive function.

Estimate gas using a Web3 provider

Ethereum nodes implement the ethestimateGas Ethereum API to predict the gas required for a transaction's successful execution. To estimate the gas usage of the ccipReceive function, you can directly call the ethestimateGas API via a Web3 provider or leverage a library like ethers.js, simplifying this interaction. This guide focuses on the ethers.js estimateGas function for gas estimation. To estimate the gas usage, execute the following command in your terminal:

```
bash
npm run estimate-gas-provider
```

Example output:

```
text
$ npm run estimate-gas-provider

> offchain-simulator@1.0.0 estimate-gas-provider
> ts-node src/estimateGasProvider.ts
```

Final Gas Usage Report:

```
Number of iterations 0 - Gas used: 5377
Number of iterations 50 - Gas used: 14946
Number of iterations 99 - Gas used: 24324
```

The estimate may exceed the actual gas used by the transaction for various reasons, including differences in node performance and EVM mechanics. For a more precise estimation, consider using Tenderly (see the next section for details).

Estimate gas using Tenderly

To estimate the gas usage of the ccipReceive function using Tenderly, execute the following command:

```
bash
npm run estimate-gas-tenderly
```

Example output:

```
text
$ npm run estimate-gas-tenderly

> offchain-simulator@1.0.0 estimate-gas-tenderly
```



```
> ts-node src/estimateGasTenderly.ts
```

Final Gas Usage Report:

Number of iterations 0 - Gas used: 5031
Number of iterations 50 - Gas used: 14581
Number of iterations 99 - Gas used: 23940

Comparison

The table below summarizes the gas estimations for different iterations using both Web3 provider and Tenderly:

Scenario	Number of iterations	Gas estimated (Web3 provider)	Gas estimated (Tenderly)
Baseline gas consumption	0	5377	5031
Average gas consumption	50	14946	14581
Peak gas consumption	99	24324	23940

The gas estimations from both Web3 provider and Tenderly are consistent across different iterations and align with actual testnet results. This demonstrates the accuracy and reliability of these offchain methods in estimating gas usage. However, you can notice that Tenderly provides more accurate results.

Conclusion

This tutorial has guided you through estimating the gas limit for the `ccipReceive` function using various methods. You have learned how to estimate gas usage in a local environment using Hardhat and Foundry, validate these estimations on testnets, and use offchain methods to estimate gas usage.

As we have explored various methods for estimating gas for the `ccipReceive` function, it is crucial to apply this knowledge effectively. Here are some targeted recommendations to enhance your approach to gas estimation:

1. Comprehensive Testing: Emphasize testing under diverse scenarios to ensure your gas estimations are robust. Different conditions can significantly affect gas usage, so covering as many cases as possible in your tests is crucial.

1. Preliminary Local Estimates: Local testing is a critical first step for estimating gas and ensuring your contracts function correctly under various scenarios. While Hardhat and Foundry facilitate development and testing, it's key to remember that these environments may not perfectly mirror your target blockchain's conditions. These initial estimates serve as valuable insights, guiding you toward more accurate gas limit settings when you proceed to testnet validations. Incorporating a buffer based on these preliminary results can help mitigate the risks of underestimating gas requirements due to environmental differences.

1. Efficiency with Offchain Methods: Since testing across different blockchains can be resource-intensive, leaning on offchain methods for gas estimation is invaluable. Tools such as Tenderly not only facilitate rapid and accurate gas usage insights on your target blockchains but also enable you to simulate the execution of the `ccipReceive` function on actual contracts deployed on mainnets. If Tenderly doesn't support a particular blockchain, a practical alternative is to use a Web3 provider that does support that chain, as illustrated in the Estimate gas using a Web3 provider section. This is particularly helpful when considering the diversity in gas metering rules across blockchains. This approach saves time and enhances the precision of your gas limit estimations, allowing for more cost-effective transactions from your dApp.

```
# cross-chain-tokens-from-eoa.mdx:
```

```
---
```

```
section: ccip
date: Last Modified
title: "Transfer Tokens between EOAs"
whatsnext: { "Checking CCIP Message Status Off-Chain": "/ccip/tutorials/get-status-offchain" }
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"
```

In this tutorial, you will use Chainlink CCIP to transfer tokens directly from your EOA (Externally Owned Account) to an account on a different blockchain. First, you will pay for CCIP fees on the source blockchain using LINK. Then, you will run the same example paying for CCIP fees in native gas, such as ETH on Ethereum or AVAX on Avalanche.

```
<Aside type="caution" title="Transferring tokens">
```

This tutorial uses the term "transferring tokens" even though the tokens are not technically transferred. Instead,

they are locked or burned on the source chain and then unlocked or minted on the destination chain. Read the Token

Pools section to understand the various mechanisms that are used to transfer value

across chains.

```
</Aside>
```

Before you begin

1. Install Node.js 18. Optionally, you can use the nvm package to switch between Node.js versions with nvm use 18.

```
shell
node -v
```

```
shell
$ node -v
v18.7.0
```

1. Your EOA (Externally Owned Account) must have both AVAX and LINK tokens on Avalanche Fuji to pay for the gas fees and CCIP fees.

- Configure MetaMask to use LINK tokens
- Acquire testnet AVAX and LINK from faucets.chain.link/fuji

1. Check the Supported Networks page to confirm that the tokens you will transfer are supported for your lane. In this example, you will transfer tokens from Avalanche Fuji to Ethereum Sepolia so check the list of supported tokens [here](#).

1. Learn how to acquire CCIP test tokens. After following this guide, your EOA (Externally Owned Account) should have CCIP-BnM tokens, and CCIP-BnM should appear in the list of your tokens in MetaMask.

1. In a terminal, clone the smart-contract-examples repository and change to the smart-contract-examples/ccip/offchain/javascript directory.

```
shell
git clone https://github.com/smartcontractkit/smart-contract-examples.git &&
```

```
\
```

```
cd smart-contract-examples/ccip/offchain/javascript
```

1. Run `npm install` to install the dependencies.

```
shell
npm install
```

1. For higher security, the examples repository imports `@chainlink/env-enc`. Use this tool to encrypt your environment variables at rest.

1. Set an encryption password for your environment variables.

```
shell
npx env-enc set-pw
```

1. Run `npx env-enc set` to configure a `.env.enc` file with the basic variables that you need to send your requests to Ethereum Sepolia.

- `AVALANCHEFUJIRPCURL`: Set a URL for the Avalanche Fuji testnet. You can sign up for a personal endpoint from Alchemy, Infura, or another node provider service.

- `ETHEREUMSEPOLIARPCURL`: Set a URL for the Ethereum Sepolia testnet. You can sign up for a personal endpoint from Alchemy, Infura, or another node provider service.

- `PRIVATEKEY`: Find the private key for your testnet wallet. If you use MetaMask, follow the instructions to Export a Private Key. Note: The offchain script uses your private key to sign any transactions you make such as transferring tokens.

```
shell
npx env-enc set
```

Tutorial

Transfer tokens and pay in LINK

In this example, you will transfer CCIP-BnM tokens from your EOA on Avalanche Fuji to an account on Ethereum Sepolia. The destination account could be an EOA (Externally Owned Account) or a smart contract. The example shows how to transfer CCIP-BnM tokens, but you can re-use the same example to transfer other tokens as long as they are supported for your lane.

For this example, CCIP fees are paid in LINK tokens. To learn how to pay CCIP fees in native AVAX, read the Pay in native section. To see a detailed description of the example code, read the code explanation section.

To transfer tokens and pay in LINK, use the following command:

```
node src/transfer-tokens.js sourceChain destinationChain destinationAccount
tokenAddress amount feeTokenAddress
```

The `feeTokenAddress` parameter specifies the token address for paying CCIP fees. The supported tokens for paying fees include LINK, the native gas token of the source blockchain (ETH for Ethereum), and the wrapped native gas token (WETH for Ethereum).

Complete the following steps in your terminal:

1. Send 1,000,000,000,000,000 (0.001 CCIP-BnM) from your EOA on Avalanche Fuji to another account on Ethereum Sepolia:

```
node src/transfer-tokens.js avalancheFuji ethereumSepolia YOURACCOUNT
0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4 10000000000000000
0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846
```

Command arguments:

Argument	Explanation
-----	-----
<CopyText text="node src/transfer-tokens.js" code/>	Node.js will execute the JavaScript code inside the transfer-tokens.js file.
<CopyText text="avalancheFuji" code/>	This specifies the source blockchain, in this case, Avalanche Fuji.
<CopyText text="ethereumSepolia" code/>	This specifies the destination blockchain, which is Ethereum Sepolia in this case.
YOURACCOUNT	This is the account address on the destination blockchain. You can replace this with your account address.
<CopyText text="0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4" code/>	This is the CCIP-BnM token contract address on Avalanche Fuji. The contract addresses for each network can be found on the Supported Networks page.
<CopyText text="10000000000000000" code/>	This is the amount of CCIP-BnM tokens to be transferred. In this example, 0.001 CCIP-BnM are transferred.
<CopyText text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>	Since you will pay for CCIP fees in LINK, this is the LINK token contract address on Avalanche Fuji. The LINK contract address can be found on the Link Token contracts page.

1. Once you execute the command, you should see the following logs:

```
$ node src/transfer-tokens.js avalancheFuji ethereumSepolia
0x83dC44a4C00DFf69d0A0c7c94B20b53a4933BE0A
0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4 10000000000000000
0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846
Estimated fees (LINK): 0.020020889739492
```

```
Approving router 0xF694E193200268f9a4868e4Aa017A0118C9a8177 to spend
10000000000000000 of token 0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4
```

```
Approval done. Transaction:
0x103c20b95183380aa7c04edd0cc8b5cd6137f0b36eda931bdd23e66fd0d21251
```

```
Approving router 0xF694E193200268f9a4868e4Aa017A0118C9a8177 to spend
fees 20020889739492000 of feeToken 0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846
```

Approval done. Transaction:
0x1b6737bbf12f1ba0391ae9ba38c46c72a1118f4d20767c4c67729cf3acc0ae8b

Calling the router to send 10000000000000000 of token
0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4 to account
0x83dC44a4C00DFf69d0A0c7c94B20b53a4933BE0A on destination chain ethereumSepolia
using CCIP

â€¦ 10000000000000000 of
Tokens(0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4) Sent to account
0x83dC44a4C00DFf69d0A0c7c94B20b53a4933BE0A on destination chain ethereumSepolia
using CCIP. Transaction hash
0x96b5b645f4f442131fc9466ff459e2211d408058be4d9a72a8fb057ca0f4723f - Message id
is 0x64be9f0e67af707d6203184adf30e86b3f0edd024c868ee0f2c57992d69609fd

Wait for message
0x64be9f0e67af707d6203184adf30e86b3f0edd024c868ee0f2c57992d69609fd to be
executed on the destination chain - Check the explorer
<https://ccip.chain.link/msg/0x64be9f0e67af707d6203184adf30e86b3f0edd024c868ee0f2c57992d69609fd>

...

1. Analyze the logs:

- The script communicates with the router to calculate the transaction fees required to transfer tokens, which amounts to 20,020,889,739,492,000 Juels (equivalent to 0.02 LINK).
- The script engages with the Link token contract, authorizing the router contract to spend 20,020,889,739,492,000 Juels for the fees and 1000000000000000 (0.001 CCIP-BnM) from your Externally Owned Account (EOA) balance.
- The script initiates a transaction through the router to transfer 1000000000000000 (0.001 CCIP-BnM) to your account on Ethereum Sepolia. It also returns the CCIP message ID.
- The script continuously monitors the destination blockchain (Ethereum Sepolia) to track the progress and completion of the cross-chain transaction.

1. While the script is waiting for the cross-chain transaction to proceed, open the CCIP explorer and search your cross-chain transaction using the message ID. Notice that the status is not finalized yet.

1. After several minutes (the waiting time depends on the finality of the source blockchain), the script will complete the polling process, and the following logs will be displayed:

Message 0x64be9f0e67af707d6203184adf30e86b3f0edd024c868ee0f2c57992d69609fd has not been processed yet on the destination chain. Try again in 60sec - Check the explorer
<https://ccip.chain.link/msg/0x64be9f0e67af707d6203184adf30e86b3f0edd024c868ee0f2c57992d69609fd>

â€¦Status of message
0x64be9f0e67af707d6203184adf30e86b3f0edd024c868ee0f2c57992d69609fd is SUCCESS - Check the explorer
<https://ccip.chain.link/msg/0x64be9f0e67af707d6203184adf30e86b3f0edd024c868ee0f2c57992d69609fd>

1. Open the CCIP explorer and use the message ID to find your cross-chain transaction.

<ClickToZoom

| <CopyText text="1000000000000000" code/> | This
is the amount of CCIP-BnM tokens to be transferred. In this example, 0.001 CCIP-
BnM are transferred.
|

1. After you execute the command, you should see the following logs:

```
$ node src/transfer-tokens.js avalancheFuji ethereumSepolia
0x83dC44a4C00DFf69d0A0c7c94B20b53a4933BE0A
0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4 1000000000000000
Estimated fees (native): 8013926492994666
```

```
Approving router 0xF694E193200268f9a4868e4Aa017A0118C9a8177 to spend
10000000000000000 of token 0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4
```

```
Approval done. Transaction:
0x23fd23e7df77ef6619ed108f507e85108e5e8592bc754a85b1264f8cf15e3221
```

```
Calling the router to send 10000000000000000 of token
0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4 to account
0x83dC44a4C00DFf69d0A0c7c94B20b53a4933BE0A on destination chain ethereumSepolia
using CCIP
```

```
â€¦ 10000000000000000 of
Tokens(0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4) Sent to account
0x83dC44a4C00DFf69d0A0c7c94B20b53a4933BE0A on destination chain ethereumSepolia
using CCIP. Transaction hash
0xe4b2226a55a6eb27f5e5ecf497af932578bbdc1009f412a8b7a855a5dbd00ffa - Message id
is 0x137481a149a892f9b555d9f0c934b67fd85354af0292b82eff2f0eafd8686b9d
```

```
Wait for message
0x137481a149a892f9b555d9f0c934b67fd85354af0292b82eff2f0eafd8686b9d to be
executed on the destination chain - Check the explorer
https://ccip.chain.link/msg/0x137481a149a892f9b555d9f0c934b67fd85354af0292b82eff
2f0eafd8686b9d
```

...

1. Analyze the logs:

- The script communicates with the router to calculate the transaction fees required to transfer tokens, which amounts to 80,139,264,929,946,666 wei (equivalent to 0.08 AVAX).
- The script interacts with the CCIP-BnM token contract, authorizing the router contract to deduct 0.001 CCIP-BnM from your Externally Owned Account (EOA) balance.
- The script initiates a transaction through the router to transfer 0.001 CCIP-BnM tokens to your destination account on Ethereum Sepolia. It also returns the CCIP message ID.
- The script continuously monitors the destination blockchain (Ethereum Sepolia) to track the progress and completion of the cross-chain transaction.

1. The transaction time depends on the finality of the source blockchain. After several minutes, the script will complete the polling process and the following logs will be displayed:

```
Message 0x137481a149a892f9b555d9f0c934b67fd85354af0292b82eff2f0eafd8686b9d
is not processed yet on destination chain.Try again in 60sec - Check the
explorer
https://ccip.chain.link/msg/0x137481a149a892f9b555d9f0c934b67fd85354af0292b82eff
2f0eafd8686b9d
```

â€¦Status of message
0x137481a149a892f9b555d9f0c934b67fd85354af0292b82eff2f0eafd8686b9d is SUCCESS -
Check the explorer
<https://ccip.chain.link/msg/0x137481a149a892f9b555d9f0c934b67fd85354af0292b82eff2f0eafd8686b9d>

1. Open the CCIP explorer and use the message ID to find your cross-chain transaction.

```
<ClickToZoom  
  src="/images/ccip/tutorials/ccip-explorer-offchain-send-tokens-tx-  
details.jpg"  
  alt="Chainlink CCIP Explorer transaction details"  
>
```

1. The data field is empty because only tokens are transferred. The gas limit is set to 0 because the transaction is directed to an Externally Owned Account (EOA), so no function calls are expected on the destination chain.

Code explanation

The Javascript featured in this tutorial is designed to interact with CCIP to transfer tokens. The contract code includes several code comments to clarify each step, but this section explains the key elements.

Imports

The script starts by importing the necessary modules and data. It imports ethers.js and ABIs (Application Binary Interface) from a config file for different contracts and configurations.

Handling arguments

The handleArguments function validates and parses the command line arguments passed to the script.

Main function: transferTokens

This asynchronous function, transferTokens performs the token transfer.

Initialization

The script initializes ethers providers to communicate with the blockchains in this section. It parses source and destination router addresses and blockchain selectors. A signer is created to sign transactions.

Token validity check

The script fetches a list of supported tokens for the destination chain and checks if the token you want to transfer is supported.

Building the CCIP message

A Cross-Chain Interoperability Protocol (CCIP) message is built, which will be sent to the router contract. It includes the destination account, amount, token address, and additional parameters.

Fee calculation

The script calls the router to estimate the fees for transferring tokens.

Transferring tokens

This section handles the actual transferring of tokens. It covers three cases:

- Fees paid using the native gas token: The contract makes one approval for the transfer amount. The fees are included in the value transaction field.
- Fees paid using an asset different from the native gas token and the token being transferred: The contracts makes two approvals. The first approval is for the transfer amount and the second approval is for the fees.
- Fees paid using the same asset that is being transferred, but not the native gas token: The contract makes a single approval for the sum of the transfer amount and fees.

The script waits for the transaction to be validated and stores the transaction receipt.

Fetching message ID

The router's `ccipSend` function returns a message ID. The script simulates a call to the blockchain to fetch the message ID that the router returned.

Checking the status on the destination chain

The script polls the off-ramp contracts on the destination chain to wait for the message to be executed. If the message is executed, it returns the status. Otherwise, the message times out after 40 minutes.

```
# cross-chain-tokens.mdx:
```

```
---
section: ccip
date: Last Modified
title: "Transfer Tokens"
whatsnext:
  {
    "Learn how to transfer tokens and send data in a single CCIP transaction":
"/ccip/tutorials/programmable-token-transfers",
    "See example cross-chain dApps and tools": "/ccip/examples",
    "See the list of supported networks": "/ccip/supported-networks",
    "Learn CCIP best practices": "/ccip/best-practices",
  }
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

In this tutorial, you will use Chainlink CCIP to transfer tokens from a smart contract to an account on a different blockchain. First, you will pay for the CCIP fees on the source blockchain using LINK. Then, you will use the same contract to pay CCIP fees in native gas tokens. For example, you would use ETH on Ethereum or AVAX on Avalanche.

```
<Aside type="note" title="Node Operator Rewards">
  CCIP rewards the oracle node and Risk Management node operators in LINK.
</Aside>
```

```
<Aside type="caution" title="Transferring tokens">
  This tutorial uses the term "transferring tokens" even though the tokens are
  not technically transferred. Instead,
  they are locked or burned on the source chain and then unlocked or minted on
  the destination chain. Read the Token
  Pools section to understand the various mechanisms that are used to transfer
  value
  across chains.
</Aside>
```

</Aside>

Before you begin

1. You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read this tutorial, which will guide you through using the Solidity programming language, interacting with the MetaMask wallet and working within the Remix Development Environment.

1. Your account must have some AVAX and LINK tokens on Avalanche Fuji. Learn how to Acquire testnet LINK.

1. Check the Supported Networks page to confirm that the tokens you will transfer are supported for your lane. In this example, you will transfer tokens from Avalanche Fuji to Ethereum Sepolia so check the list of supported tokens here.

1. Learn how to acquire CCIP test tokens. Following this guide, you should have CCIP-BnM tokens, and CCIP-BnM should appear in the list of your tokens in MetaMask.

1. Learn how to fund your contract. This guide shows how to fund your contract in LINK, but you can use the same guide to fund your contract with any ERC20 tokens as long as they appear in the list of tokens in MetaMask.

Tutorial

<CcipCommon callout="useSimulator" />

In this tutorial, you will transfer CCIP-BnM tokens from a contract on Avalanche Fuji to an account on Ethereum Sepolia. First, you will pay CCIP fees in LINK, then you will pay CCIP fees in native gas. The destination account can be an EOA (Externally Owned Account) or a smart contract. Moreover, the example shows how to transfer CCIP-BnM tokens, but you can re-use the same example to transfer other tokens as long as they are supported for your lane.

<CodeSample src="samples/CCIP/TokenTransferor.sol" />

Deploy your contracts

To use this contract:

1. Open the contract in Remix.

1. Compile your contract.

1. Deploy and fund your sender contract on Avalanche Fuji:

1. Open MetaMask and select the Avalanche Fuji network.

1. In Remix IDE, click Deploy & Run Transactions and select Injected Provider - MetaMask from the environment list. Remix will then interact with your MetaMask wallet to communicate with Avalanche Fuji.

1. Fill in your blockchain's router and LINK contract addresses. The router address can be found on the supported networks page and the LINK contract address on the LINK token contracts page. For Avalanche Fuji, the router address is <CopyText text="0xF694E193200268f9a4868e4Aa017A0118C9a8177" code/> and the LINK contract address is <CopyText text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>.

1. Click the transact button. After you confirm the transaction, the contract address appears on the Deployed Contracts list.

Note your contract address.

1. Open MetaMask and fund your contract with CCIP-BnM tokens. You can transfer <CopyText text="0.002" code/> CCIP-BnM to your contract.

1. Enable your contract to transfer tokens to Ethereum Sepolia:

1. In Remix IDE, under Deploy & Run Transactions, open the list of functions for your smart contract deployed on Avalanche Fuji.

1. Call the allowlistDestinationChain function with <CopyText

text="16015286601757825753" code/> as the destination chain selector, and <CopyText text="true" code/> as allowed. Each chain selector is found on the supported networks page.

Transfer tokens and pay in LINK

You will transfer 0.001 CCIP-BnM. The CCIP fees for using CCIP will be paid in LINK. Read this explanation for a detailed description of the code example.

1. Open MetaMask and connect to Avalanche Fuji. Fund your contract with LINK tokens. You can transfer `<CopyText text="0.1" code/>` LINK to your contract. Note: The LINK tokens are used to pay for CCIP fees.

1. Transfer CCIP-BnM from Avalanche Fuji:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, under Deploy & Run Transactions, open the list of functions for your smart contract deployed on Avalanche Fuji.

1. Fill in the arguments of the `transferTokensPayLINK` function:

Argument	Value and Description
----------	-----------------------

|-----|

```
| \destinationChainSelector | <CopyText text="16015286601757825753"
code/> <br /> CCIP Chain identifier of the destination blockchain (Ethereum
Sepolia in this example). You can find each chain selector on the supported
networks page.
```

```

| \receiver | Your account address on Ethereum Sepolia.
<br /> The destination account address. It could be a smart contract or an EOA.
|

```

```

| \token | <CopyText
text="0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4" code/><br /> The CCIP-BnM
contract address at the source chain (Avalanche Fuji in this example). You can
find all the addresses for each supported blockchain on the supported networks
page. |

```

```
<table border="0">
<tr>
<td style="width: 60%; vertical-align: top;">
<br />The token amount (0.001 CCIP-BnM).
|
```

1. Click the transact button and confirm the transaction on MetaMask.

1. Once the transaction is successful, note the transaction hash. Here is an example of a transaction on Avalanche Fuji.

```
<Aside type="note">
```

During gas price spikes, your transaction might fail, requiring more than 0.1 LINK to proceed. If your transaction fails, fund your contract with more LINK tokens and try again.

1. Open the CCIP explorer and search your cross-chain transaction using the transaction hash.


```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-send-tokens-pay-link-tx-
details.webp"
```

```
alt="Chainlink CCIP Explorer transaction details"
/>
```

1. The CCIP transaction is completed once the status is marked as "Success". The data field is empty because you are only transferring tokens.

```
<br />
```

```
<ClickToZoom
src="/images/ccip/tutorials/ccip-explorer-send-tokens-pay-link-tx-
details-success.webp"
alt="Chainlink CCIP Explorer transaction details success"
/>
```

1. Check the receiver account on the destination chain:

1. Note the destination transaction hash from the CCIP explorer. 0x083fc1a79ffcfcd617426fd71dff87ca16db2e4333e62a28cdd13d4bec0926bcb in this example.

1. Open the block explorer for your destination chain. For Ethereum Sepolia, open etherscan.

1. Search the transaction hash.

```
<br />
```

```
<ClickToZoom
src="/images/ccip/tutorials/send-tokens-pay-link-sepolia-tokens-
received.webp"
alt="Chainlink CCIP Sepolia tokens received"
/>
```

1. Notice in the Tokens Transferred section that CCIP-BnM tokens have been transferred to your account (0.001 CCIP-BnM).

Transfer tokens and pay in native

You will transfer 0.001 CCIP-BnM. The CCIP fees for using CCIP will be paid in Avalanche Fuji's native AVAX. Read this explanation for a detailed description of the code example.

1. Open MetaMask and connect to Avalanche Fuji. Fund your contract with native gas tokens. You can transfer `<CopyText text="0.1" code/>` AVAX to your contract. Note: The native gas tokens are used to pay for CCIP fees.

1. Transfer CCIP-BnM from Avalanche Fuji:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Avalanche Fuji.

1. Fill in the arguments of the transferTokensPayNative function:

```
<br />
```

Argument	Value and Description

```
-----
-----
-----
-----
```

```
| \destinationChainSelector | <CopyText text="16015286601757825753"
code/> <br /> CCIP Chain identifier of the destination blockchain (Ethereum
Sepolia in this example). You can find each chain selector on the supported
```

networks page.

\receiver	Your account address on Ethereum Sepolia.
-----------	---

 The destination account address. It could be a smart contract or an EOA.

\token	<CopyText
--------	-----------

text="0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4" code/>
 The CCIP-BnM contract address at the source chain (Avalanche Fuji in this example). You can find all the addresses for each supported blockchain on the supported networks page..

\amount	<CopyText text="10000000000000000" code/>
---------	---

 The token amount (0.001 CCIP-BnM).

1. Click the transact button and confirm the transaction on MetaMask.
1. Once the transaction is successful, note the transaction hash. Here is an example of a transaction on Avalanche Fuji.

<Aside type="note">
 During gas price spikes, your transaction might fail, requiring more than 0.01 ETH to proceed. If your transaction fails, fund your contract with more ETH and try again.
 </Aside>

1. Open the CCIP explorer and search your cross-chain transaction using the transaction hash.

<ClickToZoom
 src="/images/ccip/tutorials/ccip-explorer-send-tokens-tx-details.webp"
 alt="Chainlink CCIP Explorer transaction details"
 />

1. The CCIP transaction is completed once the status is marked as "Success". The data field is empty because you only transfer tokens. Note that CCIP fees are denominated in LINK. Even if CCIP fees are paid using native gas tokens, node operators will be paid in LINK.

<ClickToZoom
 src="/images/ccip/tutorials/ccip-explorer-send-tokens-tx-details-success.webp"
 alt="Chainlink CCIP Explorer transaction details success"
 />

1. Check the receiver account on the destination chain:

1. Note the destination transaction hash from the CCIP explorer. 0xf403d828fa377d657af67f12e99ff435974299c27ba2d57c53494d29bbbfc938 in this example.
1. Open the block explorer for your destination chain. For Ethereum Sepolia, open etherscan.
1. Search the transaction hash.

<ClickToZoom
 src="/images/ccip/tutorials/sepolia-tokens-received.webp"
 alt="Chainlink CCIP Sepolia tokens received"
 />

1. Notice in the Tokens Transferred section that CCIP-BnM tokens have been

transferred to your account (0.001 CCIP-BnM).

Explanation

```
<CcipCommon callout="importCCIPPackage" />
```

The smart contract featured in this tutorial is designed to interact with CCIP to transfer a supported token to an account on a destination chain. The contract code contains supporting comments clarifying the functions, events, and underlying logic. This section further explains initializing the contract and transferring tokens.

Initializing of the contract

When you deploy the contract, you define the router address and LINK contract address of the blockchain where you deploy the contract. The contract uses the router address to interact with the router to estimate the CCIP fees and the transmission of CCIP messages.

Transferring tokens and pay in LINK

The `transferTokensPayLINK` function undertakes six primary operations:

1. Call the `buildCCIPMessage` private function to construct a CCIP-compatible message using the `EVM2AnyMessage` struct:

- The receiver address is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through `abi.encode`.

- The data is empty because you only transfer tokens.

- The `tokenAmounts` is an array, with each element comprising a `EVMTokenAmount` struct that contains the token address and amount. The array contains one element where the token (token address) and amount (token amount) are passed by the user when calling the `transferTokensPayLINK` function.

- The `extraArgs` specifies the `gasLimit` for relaying the message to the recipient contract on the destination blockchain. In this example, the `gasLimit` is set to 0 because the contract only transfers tokens and does not expect function calls on the destination blockchain.

- The `feeTokenAddress` designates the token address used for CCIP fees. Here, `address(linkToken)` signifies payment in LINK.

```
{ " " }
```

```
<CcipCommon callout="extraArgsCallout" />
```

1. Computes the fees by invoking the router's `getFee` function.

1. Ensures your contract balance in LINK is enough to cover the fees.

1. Grants the router contract permission to deduct the fees from the contract's LINK balance.

1. Grants the router contract permission to deduct the amount from the contract's CCIP-BnM balance.

1. Dispatches the CCIP message to the destination chain by executing the router's `ccipSend` function.

Note: As a security measure, the `transferTokensPayLINK` function is protected by the `onlyAllowlistedChain` to ensure the contract owner has allowlisted a destination chain.

Transferring tokens and pay in native

The `transferTokensPayNative` function undertakes five primary operations:

1. Call the `buildCCIPMessage` private function to construct a CCIP-compatible message using the `EVM2AnyMessage` struct:

- The receiver address is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through `abi.encode`.
- The data is empty because you only transfer tokens.
- The `tokenAmounts` is an array, with each element comprising an `EVMTokenAmount` struct containing the token address and amount. The array contains one element where the token (token address) and amount (token amount) are passed by the user when calling the `transferTokensPayNative` function.
- The `extraArgs` specifies the `gasLimit` for relaying the message to the recipient contract on the destination blockchain. In this example, the `gasLimit` is set to 0 because the contract only transfers tokens and does not expect function calls on the destination blockchain.
- The `feeTokenAddress` designates the token address used for CCIP fees. Here, `address(0)` signifies payment in native gas tokens (ETH).

```
{ " " }
```

```
<CcipCommon callout="extraArgsCallout" />
```

1. Computes the fees by invoking the router's `getFee` function.
1. Ensures your contract balance in native gas is enough to cover the fees.
1. Grants the router contract permission to deduct the amount from the contract's CCIP-BnM balance.
1. Dispatches the CCIP message to the destination chain by executing the router's `ccipSend` function. Note: `msg.value` is set because you pay in native gas.

Note: As a security measure, the `transferTokensPayNative` function is protected by the `onlyAllowlistedChain`, ensuring the contract owner has allowlisted a destination chain.

```
# get-status-offchain.mdx:
```

```
---
section: ccip
date: Last Modified
title: "Checking CCIP Message Status"
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"
```

In this tutorial, you will learn how to verify the status of a Chainlink CCIP transaction offchain using JavaScript. Starting with a CCIP message ID, you'll execute the script to query the current status of a cross-chain message.

Before you begin

1. Initiate a CCIP transaction and note the CCIP message ID. You can obtain the CCIP message ID by running any of the previous CCIP tutorials.
1. Complete the prerequisites steps of the Transfer Tokens between EOAs tutorial.

Tutorial

This tutorial shows you on how to check the status of a Chainlink CCIP transaction using the `get-status.js` script. By supplying the script with the source, destination chains, and your CCIP message ID, you can verify the current status of your cross-chain message.

Execute the script in your command line:

```
bash
```

```
node src/get-status.js sourceChain destinationChain messageID
```

The script requires the following parameters:

- sourceChain is the identifier for the source blockchain. For example, avalancheFuji.
- destinationChain is the identifier for the destination blockchain. For example, ethereumSepolia.
- messageID is the unique identifier for the CCIP transaction message that you need to check.

Example Usage:

If you initiated a transaction from the Avalanche Fuji testnet to the Ethereum Sepolia testnet and received a message ID, you can check the status of this message with the following command:

```
text
$ node src/get-status.js avalancheFuji ethereumSepolia
0x25d18c6adfc1f99514b40f9931a14ca08228cdbabfc5226c1e6a43ce7441595d
```

```
Status of message
0x25d18c6adfc1f99514b40f9931a14ca08228cdbabfc5226c1e6a43ce7441595d on offRamp
0x000b26f604eAadC3D874a4404bde6D64a97d95ca is SUCCESS
```

Code Explanation

The JavaScript get-status.js is designed to check the status of a user-provided CCIP message. The contract code includes several code comments to clarify each step, but this section explains the key elements.

Imports

The script imports the required modules and data:

- Ethers.js: JavaScript library for interacting with the Ethereum Blockchain and its ecosystem.
- Router and OffRamp Contract ABIs: These Application Binary Interfaces (ABIs) enable the script to interact with specific smart contracts on the blockchain.
- Configuration Functions: Includes getProviderRpcUrl for retrieving the RPC URL of a blockchain, getRouterConfig for accessing the router smart contract's configuration, and getMessageStatus for translating numeric status codes into readable strings.

Understanding the getMessageStatus function

Before diving into the script execution, it's crucial to understand how the getMessageStatus function works. This function is designed to translate the numeric status codes returned by Solidity enums into human-readable statuses so they are clear to developers and users. The function uses a mapping defined in messageState.json, which correlates to the MessageExecutionState enum used by the Chainlink CCIP's OffRamp contract.

Handling Arguments

The handleArguments function ensures the script operates with the correct parameters. It validates the presence of three command-line arguments â the source chain identifier, the destination chain identifier, and the message ID.

Main Function: getStatus

The script's core is encapsulated in the getStatus asynchronous function. This

function completes initialization, configuration retrieval, and contract instantiation.

Initialization

Firstly, it establishes connections to the source and destination blockchain networks using the `JsonRpcProvider`.

Configuration Retrieval

The script then retrieves the configuration for router contracts on both the source and destination chains. This includes the router addresses and chain selectors.

Contract Instantiation

The script instantiates the source and destination router contracts using ethers and the router contract addresses.

Status Query

To query the status of the provided CCIP message ID, the script completes the following steps:

1. Check if the source chain's router supports the destination chain
2. Fetch OffRamp contracts associated with the destination router
3. Filter these contracts to find those that match the source chain
4. Query each matching OffRamp contract for an event related to the message ID

If an event is found, the script reads the status from the arguments. It translates the numeric status into a human-readable status and logs this information.

```
# index.mdx:
```

```
---
section: ccip
date: Last Modified
title: "CCIP tutorials"
isIndex: true
---
```

You can explore several comprehensive guides to learn about cross-chain interoperability using CCIP. These tutorials provide step-by-step instructions to help you understand different patterns that you can incorporate into your blockchain projects.

Guides

- Transfer Tokens
- Transfer Tokens with Data
- Transfer Tokens with Data - Defensive Example
- Test CCIP Locally
- Offchain
 - Transfer Tokens between EOAs
 - Checking CCIP Message Status
- Transfer USDC with Data
- Send Arbitrary Data
- Send Arbitrary Data and Receive Transfer Confirmation: A -> B -> A
- Manual Execution
- Optimizing Gas Limit Settings in CCIP Messages
- Acquire Test Tokens

```
# manual-execution.mdx:
```

```
---
section: ccip
date: Last Modified
title: "Manual Execution"
whatsnext:
  {
    "Learn how to handle errors gracefully when making CCIP transactions":
"/ccip/tutorials/programmable-token-transfers-defensive",
    "See example cross-chain dApps and tools": "/ccip/examples",
    "See the list of supported networks": "/ccip/supported-networks",
    "Learn CCIP best practices": "/ccip/best-practices",
  }
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<Aside type="note">
  Read the CCIP manual execution conceptual page to understand how manual
  execution
  works under the hood.
</Aside>
```

This tutorial is similar to the programmable token transfers example. It demonstrates the use of Chainlink CCIP for transferring tokens and arbitrary data between smart contracts on different blockchains. A distinctive feature of this tutorial is that we intentionally set a very low gas limit when using CCIP to send our message. This low gas limit is designed to cause the execution on the destination chain to fail, providing an opportunity to demonstrate the manual execution feature. Here's how you will proceed:

1. Initiate a Transfer: You'll transfer tokens and arbitrary data from your source contract on Avalanche Fuji to a receiver contract on Ethereum Sepolia. You will notice that the CCIP message has a very low gas limit, causing the execution on the receiver contract to fail.
1. Failure of CCIP Message Delivery: Once the transaction is finalized on the source chain (Avalanche Fuji), CCIP will deliver your message to the receiver contract on the destination chain (Ethereum Sepolia). You can follow the progress of your transaction using the CCIP explorer. Here, you'll observe that the execution on the receiver contract failed due to the low gas limit.
1. Manual Execution via CCIP Explorer: Using the CCIP explorer, you will override the previously set gas limit and retry the execution. This process is referred to as manual execution.
1. Confirm Successful Execution: After manually executing the transaction with an adequate gas limit, you'll see that the status of your CCIP message is updated to successful. This indicates that the tokens and data were correctly transferred to the receiver contract.

Before you begin

1. You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read this tutorial, which will guide you through using the Solidity programming language, interacting with the MetaMask wallet and working within the Remix Development Environment.
1. Your account must have some AVAX and LINK tokens on Avalanche Fuji and ETH tokens on Ethereum Sepolia. Learn how to Acquire testnet LINK.
1. Check the Supported Networks page to confirm that the tokens you will transfer are supported for your lane. In this example, you will transfer tokens from Avalanche Fuji to Ethereum Sepolia so check the list of supported tokens here.
1. Learn how to acquire CCIP test tokens. Following this guide, you should have

CCIP-BnM tokens, and CCIP-BnM should appear in the list of your tokens in MetaMask.

1. Learn how to fund your contract. This guide shows how to fund your contract in LINK, but you can use the same guide for funding your contract with any ERC20 tokens as long as they appear in the list of tokens in MetaMask.

1. Follow the previous tutorial: Transfer Tokens with Data to learn how to make programmable token transfers using CCIP.

1. Create a free account on tenderly. You will use tenderly to investigate the failed execution of the receiver contract.

Tutorial

In this tutorial, you'll send a text string and CCIP-BnM tokens between smart contracts on Avalanche Fuji and Ethereum Sepolia using CCIP and pay transaction fees in LINK. The tutorial demonstrates setting a deliberately low gas limit in the CCIP message, causing initial execution failure on the receiver contract. You will then:

1. Use the CCIP explorer to increase the gas limit.
1. Manually retry the execution.
1. Observe successful execution after the gas limit adjustment.

```
<CodeSample src="samples/CCIP/ProgrammableTokenTransfersLowGasLimit.sol" />
```

Deploy your contracts

To use this contract:

1. Open the contract in Remix.

1. Compile your contract.

1. Deploy, fund your sender contract on Avalanche Fuji and enable sending messages to Ethereum Sepolia:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, click on Deploy & Run Transactions and select Injected Provider - MetaMask from the environment list. Remix will then interact with your MetaMask wallet to communicate with Avalanche Fuji.

1. Fill in your blockchain's router and LINK contract addresses. The router address can be found on the supported networks page and the LINK contract address on the LINK token contracts page. For Avalanche Fuji, the router address is `<CopyText text="0xF694E193200268f9a4868e4Aa017A0118C9a8177" code/>` and the LINK contract address is `<CopyText text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>`.

1. Click the transact button. After you confirm the transaction, the contract address appears on the Deployed Contracts list.

Note your contract address.

1. Open MetaMask and fund your contract with CCIP-BnM tokens. You can transfer `<CopyText text="0.002" code/>` CCIP-BnM to your contract.

1. Open MetaMask and fund your contract with LINK tokens. You can transfer `<CopyText text="0.5" code/>` LINK to your contract. In this example, LINK is used to pay the CCIP fees.

1. Enable your contract to send CCIP messages to Ethereum Sepolia:

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Avalanche Fuji.

1. Call the `allowlistDestinationChain` with `<CopyText text="16015286601757825753" code/>` as the destination chain selector, and `<CopyText text="true" code/>` as allowed. Each chain selector is found on the supported networks page.

1. Deploy your receiver contract on Ethereum Sepolia and enable receiving messages from your sender contract:

1. Open MetaMask and select the network Ethereum Sepolia.

1. In Remix IDE, under Deploy & Run Transactions, make sure the environment is still Injected Provider - MetaMask.

1. Fill in your blockchain's router and LINK contract addresses. The router address can be found on the supported networks page and the LINK contract address on the LINK token contracts page. For Ethereum Sepolia, the router address is `<CopyText text="0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59" code/>` and the LINK contract address is `<CopyText text="0x779877A7B0D9E8603169DdbD7836e478b4624789" code/>`.

1. Click the transact button. After you confirm the transaction, the contract address appears on the Deployed Contracts list.

Note your contract address.

1. Enable your contract to receive CCIP messages from Avalanche Fuji:

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Ethereum Sepolia.

1. Call the `allowlistSourceChain` with `<CopyText text="14767482510784806043" code/>` as the source chain selector, and `<CopyText text="true" code/>` as allowed. Each chain selector is found on the supported networks page.

1. Enable your contract to receive CCIP messages from the contract that you deployed on Avalanche Fuji:

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Ethereum Sepolia.

1. Call the `allowlistSender` with the contract address of the contract that you deployed on Avalanche Fuji, and `<CopyText text="true" code/>` as allowed.

At this point, you have one sender contract on Avalanche Fuji and one receiver contract on Ethereum Sepolia. As security measures, you enabled the sender contract to send CCIP messages to Ethereum Sepolia and the receiver contract to receive CCIP messages from the sender and Avalanche Fuji.

Transfer and Receive tokens and data and pay in LINK

You will transfer 0.001 CCIP-BnM and a text. The CCIP fees for using CCIP will be paid in LINK.

1. Send a string data with tokens from Avalanche Fuji:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Avalanche Fuji.

1. Fill in the arguments of the `sendMessagePayLINK` function:

`
`

Argument	Value and Description
<code>\destinationChainSelector</code>	<code><CopyText text="16015286601757825753" code/></code> CCIP Chain identifier of the destination blockchain (Ethereum Sepolia in this example). You can find each chain selector on the supported networks page.
<code>\receiver</code>	Your receiver contract address at Ethereum Sepolia. <code>
</code> The destination contract address.
<code>\text</code>	<code><CopyText text="Hello World!" code/></code> <code>
</code> Any string
<code>\token</code>	<code><CopyText text="0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4" code/></code> <code>
</code> The CCIP-BnM contract address at the source chain (Avalanche Fuji in this example). You can

Manual execution

Investigate the root cause of receiver contract execution failure

To determine if a low gas limit is causing the failure in the receiver contract's execution, consider the following methods:

- Error analysis: Examine the error description in the CCIP explorer. An error labeled ReceiverError. This may be due to an out of gas error on the destination chain. Error code: 0x, often indicates a low gas issue
- Advanced Investigation Tool: For a comprehensive analysis, employ a sophisticated tool like tenderly. Tenderly can provide detailed insights into the transaction processes, helping to pinpoint the exact cause of the failure.

To use tenderly:

1. Copy the destination transaction hash from the CCIP explorer. In this example, the destination transaction hash is 0x06cb1c7d92483e67382a932e99411c4525e2c3aca6e46498c2ba64bf7eb08aba.
1. Open tenderly and search for your transaction. You should see an interface similar to the following:

```
<br />
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-manual-
execution-tenderly1.jpg"
  alt="Chainlink CCIP Sepolia open in tenderly"
/>
```

1. Enable Full Trace then click on Reverts.

```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-manual-
execution-tenderly2.jpg"
  alt="Chainlink CCIP Sepolia open in tenderly"
/>
```

1. Notice the out of gas error in the receiver contract. In this example, the receiver contract is 0x47EAa31C9e2B1B1Ba19824BedcbE0014c15df15e.

Trigger manual execution

You will increase the gas limit and trigger manual execution:

1. In the CCIP explorer, set the Gas limit override to <CopyText text="200000" code/> then click on Trigger Manual Execution.

```
<br />
<ClickToZoom
  src="/images/ccip/tutorials/sepolia-token-override-gas-limit.jpg"
  alt="Chainlink CCIP Sepolia - override gas limit"
/>
```

1. After you confirm the transaction on Metamask, the CCIP explorer shows you a confirmation screen.

```
<br />
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-manual-
```

```

execution-confirmation.jpg"
    alt="Chainlink CCIP Sepolia - override gas limit - confirmation
screen"
  />

```

1. Click on the Close button and observe the status marked as Success.

```

<br />

<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-manual-
execution-success.jpg"
  alt="Chainlink CCIP Sepolia - override gas limit - success"
/>

```

1. Check the receiver contract on the destination chain:

1. Open MetaMask and select the network Ethereum Sepolia.
1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Ethereum Sepolia.
1. Call the getLastReceivedMessageDetails function.

```

<br />

<ClickToZoom
  src="/images/ccip/tutorials/sepolia-token-messagedetails-pay-link-
manual-success.jpg"
  alt="Chainlink CCIP Sepolia message details - success"
/>

```

1. Notice the received messageId is 0xf8dc098c832332ac59ccc73ee00b480975d8f122a2265c90a1ccc2cd52268770, the received text is Hello World!, the token address is 0xFd57b4ddBf88a4e07fF4e34C487b99af2Fe82a05 (CCIP-BnM token address on Ethereum Sepolia) and the token amount is 10000000000000000 (0.001 CCIP-BnM).

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to transfer tokens and data from Avalanche Fuji to Ethereum Sepolia and from Ethereum Sepolia back to Avalanche Fuji.

Explanation

```

<CcipCommon callout="importCCIPPackage" />

```

The smart contract used in this tutorial is configured to use CCIP for transferring and receiving tokens with data, similar to the contract in the Transfer Tokens with Data tutorial. For a detailed understanding of the contract code, refer to the code explanation section of that tutorial.

A key distinction in this tutorial is the intentional setup of a low gas limit of 20,000 for building the CCIP message. This specific gas limit setting is expected to fail the message delivery on the receiver contract in the destination chain:

```

solidity
Client.argsToBytes(
  Client.EVMExtraArgsV1({gasLimit: 20000})
)

```

offchain.mdx:

```
section: ccip
date: Last Modified
title: "CCIP Offchain tutorials"
---
```

These tutorials focus on direct interaction between Externally Owned Accounts (EOAs) and the CCIP Router.

Tutorials

- Transfer Tokens between EOAs: Learn how to transfer tokens between Externally Owned Accounts (EOAs) across different blockchains, using Chainlink CCIP.
- Checking CCIP Message Status Off-Chain: Learn how to verify the status of Chainlink CCIP messages offchain using JavaScript.

```
# programmable-token-transfers-defensive.mdx:
```

```
---
section: ccip
date: Last Modified
title: "Transfer Tokens with Data - Defensive Example"
whatsnext:
  {
    "Transfer Tokens Between EOAs": "/ccip/tutorials/cross-chain-tokens-from-
    eoa",
    "See example cross-chain dApps and tools": "/ccip/examples",
    "See the list of supported networks": "/ccip/supported-networks",
    "Learn CCIP best practices": "/ccip/best-practices",
  }
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

This tutorial extends the programmable token transfers example. It uses Chainlink CCIP to transfer tokens and arbitrary data between smart contracts on different blockchains, and focuses on defensive coding in the receiver contract. In the event of a specified error during the CCIP message reception, the contract locks the tokens. Locking the tokens allows the owner to recover and redirect them as needed. Defensive coding is crucial as it enables the recovery of locked tokens and ensures the protection of your users' assets.

```
<Aside type="caution" title="Transferring tokens">
  This tutorial uses the term "transferring tokens" even though the tokens are
  not technically transferred. Instead,
  they are locked or burned on the source chain and then unlocked or minted on
  the destination chain. Read the Token
  Pools section to understand the various mechanisms that are used to transfer
  value
  across chains.
</Aside>
```

Before you begin

1. You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read this tutorial, which will guide you through using the Solidity programming language, interacting with the MetaMask wallet and working within the Remix Development Environment.
1. Your account must have some AVAX and LINK tokens on Avalanche Fuji and ETH tokens on Ethereum Sepolia. Learn how to Acquire testnet LINK.
1. Check the Supported Networks page to confirm that the tokens you will transfer are supported for your lane. In this example, you will transfer tokens from Avalanche Fuji to Ethereum Sepolia so check the list of supported tokens

here.

1. Learn how to acquire CCIP test tokens. Following this guide, you should have CCIP-BnM tokens, and CCIP-BnM should appear in the list of your tokens in MetaMask.

1. Learn how to fund your contract. This guide shows how to fund your contract in LINK, but you can use the same guide for funding your contract with any ERC20 tokens as long as they appear in the list of tokens in MetaMask.

1. Follow the previous tutorial: Transfer Tokens with Data to learn how to make programmable token transfers using CCIP.

Tutorial

```
<CcipCommon callout="useSimulator" />
```

In this guide, you'll initiate a transaction from a smart contract on Avalanche Fuji, sending a string text and CCIP-BnM tokens to another smart contract on Ethereum Sepolia using CCIP. However, a deliberate failure in the processing logic will occur upon reaching the receiver contract. This tutorial will demonstrate a graceful error-handling approach, allowing the contract owner to recover the locked tokens.

```
<Aside type="note" title="Correctly estimate your gas limit">
```

It is crucial to thoroughly test all scenarios to accurately estimate the required gas limit, including for failure

scenarios. Be aware that the gas used to execute the error-handling logic for failure scenarios may be higher than

that for successful scenarios.

```
</Aside>
```

```
<CodeSample src="samples/CCIP/ProgrammableDefensiveTokenTransfers.sol" />
```

Deploy your contracts

To use this contract:

1. Open the contract in Remix.

1. Compile your contract.

1. Deploy, fund your sender contract on Avalanche Fuji and enable sending messages to Ethereum Sepolia:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, click on Deploy & Run Transactions and select Injected Provider - MetaMask from the environment list. Remix will then interact with your MetaMask wallet to communicate with Avalanche Fuji.

1. Fill in your blockchain's router and LINK contract addresses. The router address can be found on the supported networks page and the LINK contract address on the LINK token contracts page. For Avalanche Fuji, the router address is `<CopyText text="0xF694E193200268f9a4868e4Aa017A0118C9a8177" code/>` and the LINK contract address is `<CopyText text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>`.

1. Click the transact button. After you confirm the transaction, the contract address appears on the Deployed Contracts list.

Note your contract address.

1. Open MetaMask and fund your contract with CCIP-BnM tokens. You can transfer `<CopyText text="0.002" code/>` CCIP-BnM to your contract.

1. Enable your contract to send CCIP messages to Ethereum Sepolia:

1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Avalanche Fuji.

1. Call the `allowlistDestinationChain` with `<CopyText text="16015286601757825753" code/>` as the destination chain selector, and `<CopyText text="true" code/>` as allowed. Each chain selector is found on the supported networks page.

1. Deploy your receiver contract on Ethereum Sepolia and enable receiving messages from your sender contract:

1. Open MetaMask and select the network Ethereum Sepolia.

1. In Remix IDE, under Deploy & Run Transactions, make sure the environment is still Injected Provider - MetaMask.

1. Fill in your blockchain's router and LINK contract addresses. The router address can be found on the supported networks page and the LINK contract address on the LINK token contracts page. For Ethereum Sepolia, the router address is `<CopyText text="0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59" code/>` and the LINK contract address is `<CopyText text="0x779877A7B0D9E8603169DdbD7836e478b4624789" code/>`.

1. Click the transact button. After you confirm the transaction, the contract address appears on the Deployed Contracts list.

Note your contract address.

1. Enable your contract to receive CCIP messages from Avalanche Fuji:

1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Ethereum Sepolia.

1. Call the `allowlistSourceChain` with `<CopyText text="14767482510784806043" code/>` as the source chain selector, and `<CopyText text="true" code/>` as allowed. Each chain selector is found on the supported networks page.

1. Enable your contract to receive CCIP messages from the contract that you deployed on Avalanche Fuji:

1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Ethereum Sepolia.

1. Call the `allowlistSender` with the contract address of the contract that you deployed on Avalanche Fuji, and `<CopyText text="true" code/>` as allowed.

1. Call the `setSimRevert` function, passing `true` as a parameter, then wait for the transaction to confirm. Setting `ssimRevert` to `true` simulates a failure when processing the received message. Read the explanation section for more details.

At this point, you have one sender contract on Avalanche Fuji and one receiver contract on Ethereum Sepolia. As security measures, you enabled the sender contract to send CCIP messages to Ethereum Sepolia and the receiver contract to receive CCIP messages from the sender on Avalanche Fuji. The receiver contract cannot process the message, and therefore, instead of throwing an exception, it will lock the received tokens, enabling the owner to recover them.

Note: Another security measure enforces that only the router can call the `ccipReceive` function. Read the explanation section for more details.

Recover the locked tokens

You will transfer 0.001 CCIP-BnM and a text. The CCIP fees for using CCIP will be paid in LINK.

1. Open MetaMask and connect to Avalanche Fuji. Fund your contract with LINK tokens. You can transfer `<CopyText text="0.5" code/>` LINK to your contract. In this example, LINK is used to pay the CCIP fees.

1. Send a string data with tokens from Avalanche Fuji:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Avalanche Fuji.

1. Fill in the arguments of the `sendMessagePayLINK` function:

`
`

	Argument	Value and Description

```

-----
-----
| \destinationChainSelector | <CopyText text="16015286601757825753"
code/> <br /> CCIP Chain identifier of the destination blockchain (Ethereum
Sepolia in this example). You can find each chain selector on the supported
networks page.
| \receiver | Your receiver contract address at Ethereum
Sepolia. <br /> The destination contract address.
|
| \text | <CopyText text="Hello World!" code/><br
/>Any string
|
| \token | <CopyText
text="0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4" code/><br /> The CCIP-BnM
contract address at the source chain (Avalanche Fuji in this example). You can
find all the addresses for each supported blockchain on the supported networks
page. |
| \amount | <CopyText text="10000000000000000" code/>
<br /> The token amount (0.001 CCIP-BnM).
|

```

1. Click on transact and confirm the transaction on MetaMask.
1. After the transaction is successful, record the transaction hash. Here is an example of a transaction on Avalanche Fuji.

```

<Aside type="note">
    During gas price spikes, your transaction might fail, requiring more than
0.5 LINK to proceed. If your
    transaction fails, fund your contract with more LINK tokens and try again.
</Aside>

```

1. Open the CCIP explorer and search your cross-chain transaction using the transaction hash.

```

<br />

<ClickToZoom
    src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-pay-
link-tx-details-defensive.webp"
    alt="Chainlink CCIP Explorer transaction details"
/>

```

1. The CCIP transaction is completed once the status is marked as "Success". In this example, the CCIP message ID is 0x120367995ef71f83d64a05bd7793862afda9d04049da4cb32851934490d03ae4.

```

<br />

<ClickToZoom
    src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-pay-
link-tx-details-success-defensive.webp"
    alt="Chainlink CCIP Explorer transaction details success"
/>

```

1. Check the receiver contract on the destination chain:

1. Open MetaMask and select the network Ethereum Sepolia.
1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Ethereum Sepolia.
1. Call the getFailedMessages function with an offset of <CopyText text="0" code/> and a limit of <CopyText text="1" code/> to retrieve the first failed message.


```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-send-tokens-
lastfailedmessageids-defensive.webp"
  alt="Chainlink CCIP Fuji last failed message ids"
/>
```

1. Notice the returned values are:
0x120367995ef71f83d64a05bd7793862afda9d04049da4cb32851934490d03ae4 (the message ID) and 1 (the error code indicating failure).

1. To recover the locked tokens, call the retryFailedMessage function:

Argument	Description
messageId	The unique identifier of the failed message.
tokenReceiver	The address to which the tokens will be sent.


```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-ccip-retry-failed-
defensive.webp"
  alt="Chainlink CCIP Sepolia retry failed message id"
/>
```

1. After confirming the transaction, you can open it in a block explorer.
Notice that the locked funds were transferred to the tokenReceiver address.


```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-ccip-retry-failed-tokens-
transferred-defensive.webp"
  alt="Chainlink CCIP retry failed message - tokens transferred"
/>
```

1. Call again the getFailedMessages function with an offset of <CopyText text="0" code/> and a limit of <CopyText text="1" code/> to retrieve the first failed message. Notice that the error code is now 0, indicating that the message was resolved.


```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-send-tokens-
lastfailedmessageids-defensive-recovered.webp"
  alt="Chainlink CCIP retry failed message - tokens transferred -
recovered"
/>
```

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to transfer tokens with data from Avalanche Fuji to Ethereum Sepolia and from Ethereum Sepolia back to Avalanche Fuji.

Explanation

```
<CcipCommon callout="importCCIPPackage" />
```

The smart contract featured in this tutorial is designed to interact with CCIP to transfer and receive tokens and data. The contract code is similar to the Transfer Tokens with Data tutorial. Hence, you can refer to its code explanation. We will only explain the main differences.

Sending messages

The `sendMessagePayLINK` function is similar to the `sendMessagePayLINK` function in the Transfer Tokens with Data tutorial. The main difference is the increased gas limit to account for the additional gas required to process the error-handling logic.

Receiving and processing messages

Upon receiving a message on the destination blockchain, the `ccipReceive` function is called by the CCIP router. This function serves as the entry point to the contract for processing incoming CCIP messages, enforcing crucial security checks through the `onlyRouter`, and `onlyAllowlisted` modifiers.

Here's the step-by-step breakdown of the process:

1. Entrance through `ccipReceive`:

- The `ccipReceive` function is invoked with an `Any2EVMMessage` struct containing the message to be processed.
- Security checks ensure the call is from the authorized router, an allowlisted source chain, and an allowlisted sender.

1. Processing Message:

- `ccipReceive` calls the `processMessage` function, which is external to leverage Solidity's try/catch error handling mechanism. Note: The `onlySelf` modifier ensures that only the contract can call this function.
- Inside `processMessage`, a check is performed for a simulated revert condition using the `ssimRevert` state variable. This simulation is toggled by the `setSimRevert` function, callable only by the contract owner.
- If `ssimRevert` is false, `processMessage` calls the `ccipReceive` function for further message processing.

1. Message Processing in `ccipReceive`:

- `ccipReceive` extracts and stores various information from the message, such as the `messageId`, decoded sender address, token amounts, and data.
- It then emits a `MessageReceived` event, signaling the successful processing of the message.

1. Error Handling:

- If an error occurs during the processing (or a simulated revert is triggered), the catch block within `ccipReceive` is executed.
- The `messageId` of the failed message is added to `sfailedMessages`, and the message content is stored in `smessageContents`.
- A `MessageFailed` event is emitted, which allows for later identification and reprocessing of failed messages.

Reprocessing of failed messages

The `retryFailedMessage` function provides a mechanism to recover assets if a CCIP message processing fails. It's specifically designed to handle scenarios where message data issues prevent entire processing yet allow for token recovery:

1. Initiation:

- Only the contract owner can call this function, providing the `messageId` of the failed message and the `tokenReceiver` address for token recovery.

1. Validation:

- It checks if the message has failed using `sfailedMessages.get(messageId)`.

If not, it reverts the transaction.

1. Status Update:

- The error code for the message is updated to RESOLVED to prevent reentry and multiple retries.

1. Token Recovery:

- Retrieves the failed message content using `smessageContents[messageId]`.
- Transfers the locked tokens associated with the failed message to the specified `tokenReceiver` as an escape hatch without processing the entire message again.

1. Event Emission:

- An event `MessageRecovered` is emitted to signal the successful recovery of the tokens.

This function showcases a graceful asset recovery solution, protecting user values even when message processing encounters issues.

programmable-token-transfers.mdx:

```
---
section: ccip
date: Last Modified
title: "Transfer Tokens with Data"
whatsnext:
  {
    "Learn how to manually execute a failed CCIP transaction":
"/ccip/tutorials/manual-execution",
    "Learn how to handle errors gracefully when making CCIP transactions":
"/ccip/tutorials/programmable-token-transfers-defensive",
    "Transfer Tokens Between EOAs": "/ccip/tutorials/cross-chain-tokens-from-
eoa",
    "See example cross-chain dApps and tools": "/ccip/examples",
    "See the list of supported networks": "/ccip/supported-networks",
    "Learn CCIP best practices": "/ccip/best-practices",
  }
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

In this tutorial, you will use Chainlink CCIP to transfer tokens and arbitrary data between smart contracts on different blockchains. First, you will pay for the CCIP fees on the source blockchain using LINK. Then, you will use the same contract to pay CCIP fees in native gas tokens. For example, you would use ETH on Ethereum or AVAX on Avalanche.

```
<Aside type="note" title="Node Operator Rewards">
  CCIP rewards the oracle node and Risk Management node operators in LINK.
</Aside>
```

```
<Aside type="caution" title="Transferring tokens">
  This tutorial uses the term "transferring tokens" even though the tokens are
  not technically transferred. Instead,
  they are locked or burned on the source chain and then unlocked or minted on
  the destination chain. Read the Token
  Pools section to understand the various mechanisms that are used to transfer
  value
  across chains.
</Aside>
```

Before you begin

1. You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read this tutorial, which will guide you through using the Solidity programming language, interacting with the MetaMask wallet and working within the Remix Development Environment.
1. Your account must have some AVAX and LINK tokens on Avalanche Fuji and ETH tokens on Ethereum Sepolia. Learn how to Acquire testnet LINK.
1. Check the Supported Networks page to confirm that the tokens you will transfer are supported for your lane. In this example, you will transfer tokens from Avalanche Fuji to Ethereum Sepolia so check the list of supported tokens [here](#).
1. Learn how to acquire CCIP test tokens. Following this guide, you should have CCIP-BnM tokens, and CCIP-BnM should appear in the list of your tokens in MetaMask.
1. Learn how to fund your contract. This guide shows how to fund your contract in LINK, but you can use the same guide for funding your contract with any ERC20 tokens as long as they appear in the list of tokens in MetaMask.
1. Follow the previous tutorial: Transfer tokens.

Tutorial

```
<CcipCommon callout="useSimulator" />
```

In this tutorial, you will send a string text and CCIP-BnM tokens between smart contracts on Avalanche Fuji and Ethereum Sepolia using CCIP. First, you will pay CCIP fees in LINK, then you will pay CCIP fees in native gas.

```
<CodeSample src="samples/CCIP/ProgrammableTokenTransfers.sol" />
```

Deploy your contracts

To use this contract:

1. Open the contract in Remix.
1. Compile your contract.
1. Deploy, fund your sender contract on Avalanche Fuji and enable sending messages to Ethereum Sepolia:
 1. Open MetaMask and select the network Avalanche Fuji.
 1. In Remix IDE, click on Deploy & Run Transactions and select Injected Provider - MetaMask from the environment list. Remix will then interact with your MetaMask wallet to communicate with Avalanche Fuji.
 1. Fill in your blockchain's router and LINK contract addresses. The router address can be found on the supported networks page and the LINK contract address on the LINK token contracts page. For Avalanche Fuji, the router address is `<CopyText text="0xF694E193200268f9a4868e4Aa017A0118C9a8177" code/>` and the LINK contract address is `<CopyText text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>`.
 1. Click the transact button. After you confirm the transaction, the contract address appears on the Deployed Contracts list.
Note your contract address.
 1. Open MetaMask and fund your contract with CCIP-BnM tokens. You can transfer `<CopyText text="0.002" code/>` CCIP-BnM to your contract.
 1. Enable your contract to send CCIP messages to Ethereum Sepolia:
 1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Avalanche Fuji.
 1. Call the `allowlistDestinationChain`, setting the destination chain selector to `<CopyText text="16015286601757825753" code/>` and setting `allowed to` to `<CopyText text="true" code/>`. Each chain selector is found on the supported networks page.


```
code/> <br /> CCIP Chain identifier of the destination blockchain (Ethereum
Sepolia in this example). You can find each chain selector on the supported
networks page.
| \receiver | Your receiver contract address on Ethereum
Sepolia. <br /> The destination contract address.
|
| \text | <CopyText text="Hello World!" code/><br
/>Any string
|
| \token | <CopyText
text="0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4" code/><br /> The CCIP-BnM
contract address at the source chain (Avalanche Fuji in this example). You can
find all the addresses for each supported blockchain on the supported networks
page. |
| \amount | <CopyText text="10000000000000000" code/>
<br /> The token amount (0.001 CCIP-BnM).
|
```

1. Click on transact and confirm the transaction on MetaMask.
1. After the transaction is successful, record the transaction hash. Here is an example of a transaction on Avalanche Fuji.

```
<Aside type="note">
    During gas price spikes, your transaction might fail, requiring more than
0.1 LINK to proceed. If your
    transaction fails, fund your contract with more LINK tokens and try again.
</Aside>
```

1. Open the CCIP explorer and search your cross-chain transaction using the transaction hash.

```
<br />

<ClickToZoom
    src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-pay-
link-tx-details.webp"
    alt="Chainlink CCIP Explorer transaction details"
/>
```

1. The CCIP transaction is completed once the status is marked as "Success". In this example, the CCIP message ID is 0x99a15381125e740c43a60f03c6b011ae05a3541998ca482fb5a4814417627df8.

```
<br />

<ClickToZoom
    src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-pay-
link-tx-details-success.webp"
    alt="Chainlink CCIP Explorer transaction details success"
/>
```

1. Check the receiver contract on the destination chain:

1. Open MetaMask and select the network Ethereum Sepolia.
1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Ethereum Sepolia.
1. Call the getLastReceivedMessageDetails function.

```
<br />

<ClickToZoom
    src="/images/ccip/tutorials/sepolia-token-messagedetails-pay-
link.webp"
    alt="Chainlink CCIP Sepolia message details"
```

```
style="max-width: 70%;"  
/>
```

1. Notice the received messageId is 0x99a15381125e740c43a60f03c6b011ae05a3541998ca482fb5a4814417627df8, the received text is Hello World!, the token address is 0xFd57b4ddBf88a4e07fF4e34C487b99af2Fe82a05 (CCIP-BnM token address on Ethereum Sepolia) and the token amount is 10000000000000000 (0.001 CCIP-BnM).

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to transfer tokens with data from Avalanche Fuji to Ethereum Sepolia and from Ethereum Sepolia back to Avalanche Fuji.

Transfer and Receive tokens and data and pay in native

You will transfer 0.001 CCIP-BnM and a text. The CCIP fees for using CCIP will be paid in Avalanche's native AVAX. Read this explanation for a detailed description of the code example.

1. Open MetaMask and connect to Avalanche Fuji. Fund your contract with AVAX tokens. You can transfer <CopyText text="0.15" code/> AVAX to your contract. The native gas tokens are used to pay the CCIP fees.

1. Send a string data with tokens from Avalanche Fuji:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Avalanche Fuji.

1. Fill in the arguments of the sendMessagePayNative function:

```
<br />
```

Argument	Value and Description
\destinationChainSelector	<CopyText text="16015286601757825753" code/> CCIP Chain identifier of the destination blockchain (Ethereum Sepolia in this example). You can find each chain selector on the supported networks page.
\receiver	Your receiver contract address at Ethereum Sepolia. The destination contract address.
\text	<CopyText text="Hello World!" code/> Any string
\token	<CopyText text="0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4" code/> The CCIP-BnM contract address at the source chain (Avalanche Fuji in this example). You can find all the addresses for each supported blockchain on the supported networks page.
\amount	<CopyText text="10000000000000000" code/> The token amount (0.001 CCIP-BnM).

1. Click on transact and confirm the transaction on MetaMask.

1. Once the transaction is successful, note the transaction hash. Here is an example of a transaction on Avalanche Fuji.

```
<Aside type="note">
```

During gas price spikes, your transaction might fail, requiring more than

0.01 ETH to proceed. If your transaction fails, fund your contract with more ETH and try again.

</Aside>

1. Open the CCIP explorer and search your cross-chain transaction using the transaction hash.

<ClickToZoom
src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-tx-details.webp"
alt="Chainlink CCIP Explorer transaction details"
>

1. The CCIP transaction is completed once the status is marked as "Success". In this example, the CCIP message ID is 0x32bf96ac8b01fe3f04ffa548a3403b3105b4ed479eff407ff763b7539a1d43bd. Note that CCIP fees are denominated in LINK. Even if CCIP fees are paid using native gas tokens, node operators will be paid in LINK.

<ClickToZoom
src="/images/ccip/tutorials/ccip-explorer-send-tokens-message-tx-details-success.webp"
alt="Chainlink CCIP Explorer transaction details success"
>

1. Check the receiver contract on the destination chain:

1. Open MetaMask and select the network Ethereum Sepolia.
1. In Remix IDE, under Deploy & Run Transactions, open the list of functions of your smart contract deployed on Ethereum Sepolia.
1. Call the getLastReceivedMessageDetails function.

<ClickToZoom
src="/images/ccip/tutorials/sepolia-token-messagedetails.webp"
alt="Chainlink CCIP Sepolia message details"
style="max-width: 70%;"
>

1. Notice the received messageId is 0x32bf96ac8b01fe3f04ffa548a3403b3105b4ed479eff407ff763b7539a1d43bd, the received text is Hello World!, the token address is 0xFd57b4ddBf88a4e07fF4e34C487b99af2Fe82a05 (CCIP-BnM token address on Ethereum Sepolia) and the token amount is 10000000000000000 (0.001 CCIP-BnM).

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to transfer tokens with data from Avalanche Fuji to Ethereum Sepolia and from Ethereum Sepolia back to Avalanche Fuji.

Explanation

<CcipCommon callout="importCCIPPackage" />

The smart contract featured in this tutorial is designed to interact with CCIP to transfer and receive tokens and data. The contract code contains supporting comments clarifying the functions, events, and underlying logic. Here we will further explain initializing the contract and sending data with tokens.

Initializing the contract

When deploying the contract, we define the router address and LINK contract address of the blockchain we deploy the contract on. Defining the router address is useful for the following:

- Sender part:

- Calls the router's `getFee` function to estimate the CCIP fees.
- Calls the router's `ccipSend` function to send CCIP messages.

- Receiver part:

- The contract inherits from `CCIPReceiver`, which serves as a base contract for receiver contracts. This contract requires that child contracts implement the `ccipReceive` function. `ccipReceive` is called by the `ccipReceive` function, which ensures that only the router can deliver CCIP messages to the receiver contract.

Transferring tokens and data and pay in LINK

The `sendMessagePayLINK` function undertakes six primary operations:

1. Call the `buildCCIPMessage` private function to construct a CCIP-compatible message using the `EVM2AnyMessage` struct:

- The receiver address is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through `abi.encode`.

- The data is encoded from a string to bytes using `abi.encode`.

- The `tokenAmounts` is an array, with each element comprising an `EVMTokenAmount` struct containing the token address and amount. The array contains one element where the token (token address) and amount (token amount) are passed by the user when calling the `sendMessagePayLINK` function.

- The `extraArgs` specifies the `gasLimit` for relaying the message to the recipient contract on the destination blockchain. In this example, the `gasLimit` is set to 200000.

- The `feeTokenAddress` designates the token address used for CCIP fees. Here, `address(linkToken)` signifies payment in LINK.

```
{ " " }
```

```
<CcipCommon callout="extraArgsCallout" />
```

1. Computes the fees by invoking the router's `getFee` function.

1. Ensures your contract balance in LINK is enough to cover the fees.

1. Grants the router contract permission to deduct the fees from the contract's LINK balance.

1. Grants the router contract permission to deduct the amount from the contract's CCIP-BnM balance.

1. Dispatches the CCIP message to the destination chain by executing the router's `ccipSend` function.

Note: As a security measure, the `sendMessagePayLINK` function is protected by the `onlyAllowlistedDestinationChain`, ensuring the contract owner has allowlisted a destination chain.

Transferring tokens and data and pay in native

The `sendMessagePayNative` function undertakes five primary operations:

1. Call the `buildCCIPMessage` private function to construct a CCIP-compatible message using the `EVM2AnyMessage` struct:

- The receiver address is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through `abi.encode`.

- The data is encoded from a string to bytes using `abi.encode`.
- The `tokenAmounts` is an array, with each element comprising an `EVMTokenAmount` struct containing the token address and amount. The array contains one element where the token (token address) and amount (token amount) are passed by the user when calling the `sendMessagePayNative` function.
- The `extraArgs` specifies the `gasLimit` for relaying the message to the recipient contract on the destination blockchain. In this example, the `gasLimit` is set to 200000.
- The `feeTokenAddress` designates the token address used for CCIP fees. Here, `address(0)` signifies payment in native gas tokens (ETH).

```
{ " }
```

```
<CcipCommon callout="extraArgsCallout" />
```

1. Computes the fees by invoking the router's `getFee` function.
1. Ensures your contract balance in native gas is enough to cover the fees.
1. Grants the router contract permission to deduct the amount from the contract's CCIP-BnM balance.
1. Dispatches the CCIP message to the destination chain by executing the router's `ccipSend` function. Note: `msg.value` is set because you pay in native gas.

Note: As a security measure, the `sendMessagePayNative` function is protected by the `onlyAllowlistedDestinationChain`, ensuring the contract owner has allowlisted a destination chain.

Receiving messages

On the destination blockchain, the router invokes the `ccipReceive` function which expects a `Any2EVMMessage` struct that contains:

- The CCIP `messageId`.
- The `sourceChainSelector`.
- The sender address in bytes format. Given that the sender is known to be a contract deployed on an EVM-compatible blockchain, the address is decoded from bytes to an Ethereum address using the ABI specifications.
- The `tokenAmounts` is an array containing received tokens and their respective amounts. Given that only one token transfer is expected, the first element of the array is extracted.
- The data, which is also in bytes format. Given a string is expected, the data is decoded from bytes to a string using the ABI specifications.

Note: Three important security measures are applied:

- `ccipReceive` is called by the `ccipReceive` function, which ensures that only the router can deliver CCIP messages to the receiver contract. See the `onlyRouter` modifier for more information.
- The modifier `onlyAllowlisted` ensures that only a call from an allowlisted source chain and sender is accepted.

send-arbitrary-data-receipt-acknowledgment.mdx:

```
---
```

```
section: ccip
```

```
date: Last Modified
```

```
title: "Send Arbitrary Data and Receive Transfer Confirmation: A -> B -> A"
```

```
whatsnext:
```

```
{
```

```
  "See example cross-chain dApps and tools": "/ccip/examples",
```

```
  "See the list of supported networks": "/ccip/supported-networks",
```

```
  "Learn about CCIP Architecture and Billing": "/ccip/architecture",
```

```
  "Learn CCIP best practices": "/ccip/best-practices",
```

```
}  
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"  
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

This tutorial will teach you how to use Chainlink CCIP to send arbitrary data between smart contracts on different blockchains and how to track the status of each sent message in the sender contract on the source chain. Tracking the status of sent messages allows your smart contracts to execute actions after the receiver acknowledges it received the message. In this example, the sender contract emits an event after it receives acknowledgment from the receiver.

Note: For simplicity, this tutorial demonstrates this pattern for sending arbitrary data. However, you are not limited to this application. You can apply the same pattern to programmable token transfers.

Before you begin

- This tutorial assumes you have completed the Send Arbitrary Data tutorial.
- Your account must have some AVAX tokens on Avalanche Fuji and ETH tokens on Ethereum Sepolia.
- Learn how to Acquire testnet LINK and Fund your contract with LINK.

Tutorial

```
<CcipCommon callout="useSimulator" />
```

In this tutorial, you will deploy a message tracker contract on the source blockchain (Avalanche Fuji) and an acknowledger on the destination blockchain (Ethereum Sepolia). Throughout the tutorial, you will pay for CCIP fees using LINK tokens. Here is a step-by-step breakdown:

1. Sending and building a CCIP message: Initiate and send a message from the message tracker contract on Avalanche Fuji to the acknowledger contract on Ethereum Sepolia. The message tracker contract constructs a CCIP message that encapsulates a text string and establishes a tracking status for this message before sending it off.
1. Receiving and acknowledging the message: After the acknowledger contract receives the text on Ethereum Sepolia, it sends back a CCIP message to the message tracker contract as an acknowledgment of receipt.
1. Updating tracking status: After the message tracker receives the acknowledgment, the contract updates the tracking status of the initial CCIP message and emits an event to signal completion.

Deploy the message tracker (sender) contract

Deploy the MessageTracker.sol contract on Avalanche Fuji and enable it to send and receive CCIP messages to and from Ethereum Sepolia. You must also enable your contract to receive CCIP messages from the acknowledger contract.

1. Open the MessageTracker.sol contract in Remix.

```
<CodeSample src="samples/CCIP/MessageTracker.sol" showButtonOnly={true} />
```

Note: The contract code is also available in the Examine the code section.

1. Compile the contract.

1. Deploy the contract on Avalanche Fuji:

1. Open MetaMask and select the Avalanche Fuji network.

1. On the Deploy & Run Transactions tab in Remix, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to

communicate with Avalanche Fuji.

1. Under the Deploy section, fill in the router address and the LINK token contract address for your specific blockchain. You can find both of these addresses on the Supported Networks page. The LINK token contract address is also listed on the LINK Token Contracts page. For Avalanche Fuji:

- The router address is `<CopyText text="0xF694E193200268f9a4868e4Aa017A0118C9a8177" code/>`
- The LINK token address is `<CopyText text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>`

1. Click transact to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract on Avalanche Fuji.

1. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy your contract address.

1. Open MetaMask and send `<CopyText text="0.5" code/>` LINK to the contract address you copied. Your contract will pay CCIP fees in LINK.

1. Allow the Ethereum Sepolia chain selector for both destination and source chains.

1. On the Deploy & Run Transactions tab in Remix, expand the message tracker contract in the Deployed Contracts section.

1. Call the `allowlistDestinationChain` function with `<CopyText text="16015286601757825753" code/>` as the destination chain selector for Ethereum Sepolia and `<CopyText text="true" code />` as allowed.

1. Call the `allowlistSourceChain` function with `<CopyText text="16015286601757825753" code/>` as the source chain selector for Ethereum Sepolia and `<CopyText text="true" code />` as allowed.

You can find each network's chain selector on the supported networks page.

Deploy the acknowledger (receiver) contract

Deploy the Acknowledger.sol contract on Ethereum Sepolia and enable it to send and receive CCIP messages to and from Avalanche Fuji. You must also enable your contract to receive CCIP messages from the message tracker contract.

1. Open the Acknowledger.sol contract in Remix.

`<CodeSample src="samples/CCIP/Acknowledger.sol" showButtonOnly={true} />`

Note: The contract code is also available in the Examine the code section.

1. Compile the contract.

1. Deploy the contract on Ethereum Sepolia:

1. Open MetaMask and select the Ethereum Sepolia network.

1. On the Deploy & Run Transactions tab in Remix, make sure the Environment is still set to Injected Provider - MetaMask.

1. Under the Deploy section, fill in the router address and the LINK token contract address for your specific blockchain. You can find both of these addresses on the Supported Networks page. The LINK token contract address is also listed on the LINK Token Contracts page. For Ethereum Sepolia:

- The Router address is `<CopyText text="0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59" code/>`.
- The LINK token address is `<CopyText text="0x779877A7B0D9E8603169DdbD7836e478b4624789" code/>`.

1. Click transact to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Ethereum Sepolia.

1. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy this contract address.

1. Open MetaMask and send `<CopyText text="0.5" code/>` LINK to the contract address that you copied. Your contract will pay CCIP fees in LINK.

1. Allow the Avalanche Fuji chain selector for both destination and source chains. You must also enable your acknowledger contract to receive CCIP messages from the message tracker you deployed on Avalanche Fuji.

1. On the Deploy & Run Transactions tab in Remix, expand the acknowledger contract in the Deployed Contracts section. Expand the `allowlistDestinationChain`, `allowlistSender`, and `allowlistSourceChain` functions and fill in the following arguments:

Function	Description
Value (Avalanche Fuji)	

allowlistDestinationChain	CCIP Chain identifier of the target blockchain. You can find each network's chain selector on the supported networks page <code><CopyText text="14767482510784806043" code/></code> , <code><CopyText text="true" code/></code>
allowlistSender	The address of the message tracker contract deployed on Avalanche Fuji Your deployed contract address, <code><CopyText text="true" code/></code>
allowlistSourceChain	CCIP Chain identifier of the source blockchain. You can find each network's chain selector on the supported networks page <code><CopyText text="14767482510784806043" code/></code> , <code><CopyText text="true" code/></code>

1. Open MetaMask and select the Ethereum Sepolia network.

1. For each function you expanded and filled in the arguments for, click the transact button to call the function. MetaMask prompts you to confirm the transaction. Wait for each transaction to succeed before calling the following function.

1. Finally, enable your message tracker contract to receive CCIP messages from the acknowledger contract you deployed on Ethereum Sepolia.

1. On the Deploy & Run Transactions tab in Remix, expand the message tracker contract in the Deployed Contracts section. Expand the `allowlistSender` function and fill in your acknowledger contract address and `<CopyText text="true" code/>` as allowed.

1. Open MetaMask and select the Avalanche Fuji network.

1. Click transact to call the function. MetaMask prompts you to confirm the transaction.

At this point, you have one message tracker (sender) contract on Avalanche Fuji and one acknowledger (receiver) contract on Ethereum Sepolia. You sent 0.5 LINK to the message tracker contract and 0.5 LINK to the acknowledger contract to pay the CCIP fees.

Send data and track the message status

Initial message

1. Send a Hello World! string from your message tracker contract on Avalanche Fuji to your acknowledger contract deployed on Ethereum Sepolia. You will track the status of this message during this tutorial.

1. Open MetaMask and select the Avalanche Fuji network.

1. On the Deploy & Run Transactions tab in Remix, expand the message tracker contract in the Deployed Contracts section.

1. Expand the sendMessagePayLINK function and fill in the following arguments:

Argument	Description
Value (Ethereum Sepolia)	

destinationChainSelector	CCIP Chain identifier of the target blockchain. You can find each network's chain selector on the supported networks page
<CopyText text="16015286601757825753" code/>	
receiver	The destination smart contract address
Your deployed acknowledger contract address	
text	Any string
<CopyText text="Hello World!" code/>	

1. Click transact to call the function. MetaMask prompts you to confirm the transaction.

```
<Aside type="note">
  During gas price spikes, your transaction might fail, requiring more
  than 0.5 LINK to proceed. If your
  transaction fails, fund your contract with more LINK tokens and try
  again.
</Aside>
```

1. Upon transaction success, expand the last transaction in the Remix log and copy the transaction hash. In this example, it is 0x1f88abc33a4ab426a5466e01d9e5fe8a2b96d6a6e5cedb643a674489c74126b4.

1. Open the CCIP Explorer and use the transaction hash that you copied to search for your cross-chain transaction.

```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-fuji-sepolia.jpg"
  alt="Chainlink CCIP Explorer - Fuji to Sepolia Transaction Details"
  style="max-width: 70%;"
/>
```

After the transaction is finalized on the source chain, it will take a few minutes for CCIP to deliver the data to Ethereum Sepolia and call the ccipReceive function on your acknowledger contract.

1. Copy the message ID from the CCIP Explorer transaction details. You will use this message ID to track your message status on the message tracker contract. In this example, it is 0xdd8be2f5f5d5cf3b8640c62924025b311ae83c6144f0f2ed5c24637436d6aab8.

1. On the Deploy & Run Transactions tab in Remix, expand your message tracker contract in the Deployed Contracts section.

1. Paste the message ID you copied from the CCIP explorer as the argument in the messagesInfo getter function. Click messagesInfo to read the message status.

```
<ClickToZoom
  src="/images/ccip/tutorials/message-tracker-message-status-1.jpg"
```

```

    alt="Chainlink CCIP - Message Tracker Get Message Status - 1"
    style="max-width: 70%;"/>
/>

```

Note the returned status 1. This value indicates that the message tracker contract has updated your message status to the Sent status as defined by the MessageStatus enum in the message tracker contract.

```

solidity
// Enum is used to track the status of messages sent via CCIP.
// NotSent indicates a message has not yet been sent.
// Sent indicates that a message has been sent to the Acknowledger contract
but not yet acknowledged.
// ProcessedOnDestination indicates that the Acknowledger contract has
processed the message and that
// the Message Tracker contract has received the acknowledgment from the
Acknowledger contract.
enum MessageStatus {
    NotSent, // 0
    Sent, // 1
    ProcessedOnDestination // 2
}

```

1. When the transaction is marked with a "Success" status on the CCIP Explorer, the CCIP transaction and the destination transaction are complete. The acknowledger contract has received the message from the message tracker contract.

```

<ClickToZoom
    src="/images/ccip/tutorials/ccip-explorer-fuji-sepolia-success.jpg"
    alt="Chainlink CCIP Explorer - Fuji to Sepolia Transaction Success"
    style="max-width: 70%;"/>
/>

```

Acknowledgment message

The acknowledger contract processes the message, sends an acknowledgment message containing the initial message ID back to the message tracker contract, and emits an AcknowledgmentSent event. Read this explanation for further description.

```

solidity
// Emitted when an acknowledgment message is successfully sent back to the
sender contract.
// This event signifies that the Acknowledger contract has recognized the
receipt of an initial message
// and has informed the original sender contract by sending an acknowledgment
message,
// including the original message ID.
event AcknowledgmentSent(
    bytes32 indexed messageId, // The unique ID of the CCIP message.
    uint64 indexed destinationChainSelector, // The chain selector of the
destination chain.
    address indexed receiver, // The address of the receiver on the destination
chain.
    bytes32 data, // The data being sent back, usually containing the message ID
of the original message to acknowledge its receipt.
    address feeToken, // The token address used to pay CCIP fees for sending the
acknowledgment.
    uint256 fees // The fees paid for sending the acknowledgment message via CCIP.
);

```

1. Copy your acknowledger contract address from Remix. Open the Ethereum Sepolia explorer and search for your deployed acknowledger contract. Click the Events tab to see the events log.

```
<ClickToZoom
  src="/images/ccip/tutorials/ethereum-sepolia-explorer-acknowledger-
events.jpg"
  alt="Chainlink CCIP - Ethereum Sepolia Acknowledger Contract Events"
  style="max-width: 70%;"
/>
```

The first indexed topic (topic1) in the AcknowledgmentSent event is the acknowledgment message ID sent to the message tracker contract on Avalanche Fuji. In this example, the message ID is 0xd4d4a5d0db05dc714f8150c1af654ed34eb8c9f7547401fa9bf072a815f56ac1.

1. Copy your own message ID from the indexed topic1 and search for it in the CCIP explorer.

```
<ClickToZoom
  src="/images/ccip/tutorials/ccip-explorer-sepolia-fuji-success.jpg"
  alt="Chainlink CCIP - CCIP Explorer Sepolia to Fuji Transaction Success"
  style="max-width: 70%;"
/>
```

When the transaction is marked with a "Success" status on the CCIP explorer, the CCIP transaction and the destination transaction are complete. The message tracker contract has received the message from the acknowledger contract.

Final status check

When the message tracker receives the acknowledgment message, the ccipReceive function updates the initial message status to 2, which corresponds to the ProcessedOnDestination status as defined by the MessageStatus enum. The function emits a MessageProcessedOnDestination event.

1. Open MetaMask and select the Avalanche Fuji network.

1. On the Deploy & Run Transactions tab in Remix, expand your message tracker contract in the Deployed Contracts section.

1. Copy the initial message ID from the CCIP explorer (transaction from Avalanche Fuji to Ethereum Sepolia) and paste it as the argument in the messagesInfo getter function. Click messagesInfo to read the message status. It returns status 2 and the acknowledgment message ID that confirms this status.

```
<ClickToZoom
  src="/images/ccip/tutorials/message-tracker-message-status-2.jpg"
  alt="Chainlink CCIP - Message Tracker Get Message Status - 2"
  style="max-width: 70%;"
/>
```

1. Copy your message tracker contract address from Remix. Open the Avalanche Fuji explorer and search for your deployed message tracker contract. Then, click on the Events tab.

```
<ClickToZoom
  src="/images/ccip/tutorials/message-tracker-messageconfirmed-event.jpg"
  alt="Chainlink CCIP - Message Tracker Message Confirmed Event"
  style="max-width: 70%;"
/>
```

The MessageProcessedOnDestination event is emitted with the acknowledged message ID 0xdd8be2f5f5d5cf3b8640c62924025b311ae83c6144f0f2ed5c24637436d6aab8 as

indexed topic2.

```
solidity
// Event emitted when the sender contract receives an acknowledgment
// that the receiver contract has successfully received and processed the
message.
event MessageProcessedOnDestination(
    bytes32 indexed messageId, // The unique ID of the CCIP acknowledgment
message.
    bytes32 indexed acknowledgedMsgId, // The unique ID of the message
acknowledged by the receiver.
    uint64 indexed sourceChainSelector, // The chain selector of the source
chain.
    address sender // The address of the sender from the source chain.
);
```

Explanation

```
<CcipCommon callout="importCCIPPackage" />
```

The smart contracts featured in this tutorial are designed to interact with CCIP to send and receive messages with an acknowledgment of receipt mechanism. The contract code across both contracts contains supporting comments clarifying the functions, events, and underlying logic.

Refer to the Send Arbitrary Data tutorial for more explanation about initializing the contracts, sending data, paying in LINK, and receiving data.

Here, we will further explain the acknowledgment of receipt mechanism.

Message acknowledgment of receipt mechanism

This mechanism ensures that a message sent by the message tracker (sender) contract is received and acknowledged by the acknowledger (receiver) contract. The message status is tracked and stored in the message tracker contract.

```
solidity
// Enum is used to track the status of messages sent via CCIP.
// NotSent indicates a message has not yet been sent.
// Sent indicates that a message has been sent to the Acknowledger contract but
not yet acknowledged.
// ProcessedOnDestination indicates that the Acknowledger contract has processed
the message and that
// the Message Tracker contract has received the acknowledgment from the
Acknowledger contract.
enum MessageStatus {
    NotSent, // 0
    Sent, // 1
    ProcessedOnDestination // 2
}

// Struct to store the status and acknowledger message ID of a message.
struct MessageInfo {
    MessageStatus status;
    bytes32 acknowledgerMessageId;
}

// Mapping to keep track of message IDs to their info (status & acknowledger
message ID).
mapping(bytes32 => MessageInfo) public messagesInfo;
```

Message tracker contract

The message tracker contract acts as the sender, initiating cross-chain communication. It performs the following operations:

- Message sending: Constructs and sends messages to the acknowledger contract on another blockchain, using `sendMessagePayLINK` function. On top of its five primary operations, the `sendMessagePayLINK` function also updates the message status upon sending.

- Status tracking:

- Upon sending a message, the message tracker updates its internal state to mark the message as Sent (status 1). This status is pivotal for tracking the message lifecycle and awaiting acknowledgment.

```
solidity
// Update the message status to Sent
messagesInfo[messageId].status = MessageStatus.Sent;
```

- Upon receiving an acknowledgment message from the acknowledger contract, the message tracker contract updates the message status from Sent (status 1) to `ProcessedOnDestination` (status 2). This update indicates that the cross-chain communication cycle is complete, and the receiver successfully received and acknowledged the message.

```
solidity
// Update the message status to ProcessedOnDestination
messagesInfo[messageId].status = MessageStatus.ProcessedOnDestination;
```

Acknowledger contract

The acknowledger contract receives the message, sends back an acknowledgment message, and emits an event. It performs the following operations:

- Message receipt: Upon receiving a message via CCIP, the `ccipReceive` function decodes it and calls the `acknowledgePayLINK` function nested within the `ccipReceive` function.

- Acknowledgment sending: The `acknowledgePayLINK` function acts as a custom `sendMessagePayLINK` function nested within the `ccipReceive` function. It sends an acknowledgment (a CCIP message) to the message tracker contract upon the initial message receipt. The data transferred in this acknowledgment message is the initial message ID. It then emits an `AcknowledgmentSent` event.

Security and integrity

Both contracts use allowlists to process only messages from and to allowed sources.

- Allowlisting chains and senders:

- The `sendMessagePayLINK` function is protected by the `onlyAllowlistedDestinationChain` modifier, ensuring the contract owner has allowlisted a destination chain.

- The `ccipReceive` function is protected by the `onlyAllowlisted` modifier, ensuring the contract owner has allowlisted a source chain and a sender.

- Ensuring the initial message authenticity: The message tracker contract first checks that the message awaiting acknowledgment was sent from the contract itself and is currently marked as Sent. Once confirmed, the message status is updated to `ProcessedOnDestination`.

Examine the code

MessageTracker.sol

```
<CodeSample src="samples/CCIP/MessageTracker.sol" />
```

Acknowledger.sol

```
<CodeSample src="samples/CCIP/Acknowledger.sol" />
```

Final note

In this example, the message tracker contract emits an event when it receives the acknowledgment message confirming the initial message reception and processing on the counterpart chain. However, you could think of any other logic to execute when the message tracker receives the acknowledgment. This tutorial demonstrates the pattern for sending arbitrary data, but you can apply the same pattern to programmable token transfers.

```
# send-arbitrary-data.mdx:
```

```
---
section: ccip
date: Last Modified
title: "Send Arbitrary Data"
whatsnext:
  {
    "Send Arbitrary Data and Receive Transfer Confirmation: A -> B -> A":
"/ccip/tutorials/send-arbitrary-data-receipt-acknowledgment",
    "See example cross-chain dApps and tools": "/ccip/examples",
    "See the list of supported networks": "/ccip/supported-networks",
    "Learn about CCIP Architecture and Billing": "/ccip/architecture",
    "Learn CCIP best practices": "/ccip/best-practices",
  }
---
```

```
import { CodeSample, ClickToZoom, CopyText, Aside } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

In this tutorial, you will use Chainlink CCIP to send data between smart contracts on different blockchains. First, you will pay for the CCIP fees on the source blockchain using LINK. Then, you will use the same contract to pay CCIP fees in native gas tokens. For example, you would use ETH on Ethereum or AVAX on Avalanche.

```
<Aside type="note" title="Node Operator Rewards">
  CCIP rewards the oracle node and Risk Management node operators in LINK.
</Aside>
```

Before you begin

- You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read this tutorial, which will guide you through using the Solidity programming language, interacting with the MetaMask wallet and working within the Remix Development Environment.
- Your account must have some AVAX tokens on Avalanche Fuji and ETH tokens on Ethereum Sepolia.
- Learn how to Acquire testnet LINK and Fund your contract with LINK.

Tutorial

```
<CcipCommon callout="useSimulator" />
```

In this tutorial, you will send a string text between smart contracts on Avalanche Fuji and Ethereum Sepolia using CCIP. First, you will pay CCIP fees in LINK, then you will pay CCIP fees in native gas.

```
<CodeSample src="samples/CCIP/Messenger.sol" />
```

Deploy your contracts

To use this contract:

1. Open the contract in Remix.

1. Compile your contract.

1. Deploy your sender contract on Avalanche Fuji and enable sending messages to Ethereum Sepolia:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, click on Deploy & Run Transactions and select Injected Provider - MetaMask from the environment list. Remix will then interact with your MetaMask wallet to communicate with Avalanche Fuji.

1. Fill in the router address and the link address for your network. You can find the router address on the supported networks page and the LINK token address on the LINK Token contracts page. For Avalanche Fuji, the router address is `<CopyText text="0xF694E193200268f9a4868e4Aa017A0118C9a8177" code/>` and the LINK contract address is `<CopyText text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>`.

1. Click on transact. After you confirm the transaction, the contract address appears on the Deployed Contracts list.

- Note your contract address.

1. Enable your contract to send CCIP messages to Ethereum Sepolia:

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Avalanche Fuji.

1. Call the `allowlistDestinationChain` with `<CopyText text="16015286601757825753" code/>` as the destination chain selector, and `<CopyText text="true" code/>` as allowed. Each chain selector is found on the supported networks page.

1. Deploy your receiver contract on Ethereum Sepolia and enable receiving messages from your sender contract:

1. Open MetaMask and select the network Ethereum Sepolia.

1. In Remix IDE, under Deploy & Run Transactions, make sure the environment is still Injected Provider - MetaMask.

1. Fill in the router address and the LINK address for your network. You can find the router address on the supported networks page and the LINK contract address on the LINK token contracts page. For Ethereum Sepolia, the router address is `<CopyText text="0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59" code/>` and the LINK contract address is `<CopyText text="0x779877A7B0D9E8603169DdbD7836e478b4624789" code/>`.

1. Click on transact. After you confirm the transaction, the contract address appears on the Deployed Contracts list.

- Note your contract address.

1. Enable your contract to receive CCIP messages from Avalanche Fuji:

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Ethereum Sepolia.

1. Call the `allowlistSourceChain` with `<CopyText text="14767482510784806043" code/>` as the source chain selector, and `<CopyText text="true" code/>` as allowed. Each chain selector is found on the supported networks page.

1. Enable your contract to receive CCIP messages from the contract that you deployed on Avalanche Fuji:

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Ethereum Sepolia.

1. Call the `allowlistSender` with the contract address of the contract that

you deployed on Avalanche Fuji, and `<CopyText text="true" code/>` as allowed.

At this point, you have one sender contract on Avalanche Fuji and one receiver contract on Ethereum Sepolia. As security measures, you enabled the sender contract to send CCIP messages to Ethereum Sepolia and the receiver contract to receive CCIP messages from the sender and Avalanche Fuji. Note: Another security measure enforces that only the router can call the `ccipReceive` function. Read the explanation section for more details.

Send data and pay in LINK

You will use CCIP to send a text. The CCIP fees for using CCIP will be paid in LINK. Read this explanation for a detailed description of the code example.

1. Open MetaMask and connect to Avalanche Fuji. Fund your contract with LINK tokens. You can transfer `<CopyText text="0.5" code/>` LINK to your contract. In this example, LINK is used to pay the CCIP fees.

1. Send "Hello World!" from Avalanche Fuji:

1. Open MetaMask and select the network Avalanche Fuji.
1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Avalanche Fuji.
1. Fill in the arguments of the `sendMessagePayLINK` function:

`
`

Argument	Description
Value (Ethereum Sepolia)	

<code>\destinationChainSelector</code>	CCIP Chain identifier of the target blockchain. You can find each network's chain selector on the supported networks page <code><CopyText text="16015286601757825753" code/></code>
<code>\receiver</code>	The destination smart contract address
Your deployed receiver contract address	
<code>\text</code>	any string
<code><CopyText text="Hello World!" code/></code>	

1. Click on transact and confirm the transaction on MetaMask.
1. Once the transaction is successful, note the transaction hash. Here is an example of a transaction on Avalanche Fuji.

`<Aside type="note">`
During gas price spikes, your transaction might fail, requiring more than 0.5 LINK to proceed. If your transaction fails, fund your contract with more LINK tokens and try again.
`</Aside>`

1. Open the CCIP explorer and search your cross-chain transaction using the transaction hash.

1. The CCIP transaction is completed once the status is marked as "Success". Note: In this example, the CCIP message ID is `0x28a804fa891bde8fb4f6617931187e1033a128c014aa76465911613588bc306f`.

`
`

`<ClickToZoom`
`src="/images/ccip/tutorials/ccip-explorer-pay-link-tx-success.jpg"`
`alt="Chainlink CCIP Explorer transaction success"`
`/>`

1. Check the receiver contract on the destination chain:

1. Open MetaMask and select the network Ethereum Sepolia.
1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Ethereum Sepolia.
1. Call the `getLastReceivedMessageDetails`.


```
<ClickToZoom
  src="/images/ccip/tutorials/sepolia-getmessagedetails-pay-link.jpg"
  alt="Chainlink CCIP Sepolia message details"
/>
```

1. Notice the received text is the one you sent, "Hello World!" and the message ID is the one you expect
0x28a804fa891bde8fb4f6617931187e1033a128c014aa76465911613588bc306f.

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to send data from Avalanche Fuji to Ethereum Sepolia and from Ethereum Sepolia back to Avalanche Fuji.

Send data and pay in native

You will use CCIP to send a text. The CCIP fees for using CCIP will be paid in native gas. Read this explanation for a detailed description of the code example.

1. Open MetaMask and connect to Avalanche Fuji. Fund your contract with AVAX. You can transfer `<CopyText text="1" code/>` AVAX to your contract. In this example, AVAX is used to pay the CCIP fees.

1. Send "Hello World!" from Avalanche Fuji:

1. Open MetaMask and select the network Avalanche Fuji.
1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Avalanche Fuji.
1. Fill in the arguments of the `sendMessagePayNative` function:

	Argument	Description
Value (Ethereum Sepolia)		

	<code>\destinationChainSelector</code>	CCIP Chain identifier of the target blockchain. You can find each network's chain selector on the supported networks page <code><CopyText text="16015286601757825753" code/></code>
	<code>\receiver</code>	The destination smart contract address
Your deployed receiver contract address		
	<code>\text</code>	any string
<code><CopyText text="Hello World!" code/></code>		

1. Click on transact and confirm the transaction on MetaMask.
1. Once the transaction is successful, note the transaction hash. Here is an example of a transaction on Avalanche Fuji.

<Aside type="note">

During gas price spikes, your transaction might fail, requiring more than 0.01 ETH to proceed. If your transaction fails, fund your contract with more ETH and try again.

</Aside>

1. Open the CCIP explorer and search your cross-chain transaction using the transaction hash.

<ClickToZoom
src="/images/ccip/tutorials/ccip-explorer-pay-native-tx-details.jpg"
alt="Chainlink CCIP Explorer transaction details"
>

1. The CCIP transaction is completed once the status is marked as "Success". In this example, the CCIP message ID is 0xb8cb414128f440e115dcd5d6ead50e14d250f9a47577c38af4f70deb14191457. Note that CCIP fees are denominated in LINK. Even if CCIP fees are paid using native gas tokens, node operators will be paid in LINK.

<ClickToZoom
src="/images/ccip/tutorials/ccip-explorer-pay-native-tx-success.jpg"
alt="Chainlink CCIP Explorer transaction success"
>

1. Check the receiver contract on the destination chain:

1. Open MetaMask and select the network Ethereum Sepolia.
1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Ethereum Sepolia.
1. Call the getLastReceivedMessageDetails.

<ClickToZoom
src="/images/ccip/tutorials/sepolia-getmessagedetails-pay-native.jpg"
alt="Chainlink CCIP Sepolia message details"
>

1. Notice the received text is the one you sent, "Hello World!" and the message ID is the one you expect 0xb8cb414128f440e115dcd5d6ead50e14d250f9a47577c38af4f70deb14191457.

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to send data from Avalanche Fuji to Ethereum Sepolia and from Ethereum Sepolia back to Avalanche Fuji.

Explanation

<CcipCommon callout="importCCIPPackage" />

The smart contract featured in this tutorial is designed to interact with CCIP to send and receive messages. The contract code contains supporting comments clarifying the functions, events, and underlying logic. Here we will further explain initializing the contract and sending and receiving data.

Initializing of the contract

When deploying the contract, we define the router address and LINK contract address of the blockchain we deploy the contract on. Defining the router address is useful for the following:

- Sender part:

- Calls the router's `getFee` function to estimate the CCIP fees.
 - Calls the router's `ccipSend` function to send CCIP messages.
- Receiver part:
- The contract inherits from `CCIPReceiver`, which serves as a base contract for receiver contracts. This contract requires that child contracts implement the `ccipReceive` function. `ccipReceive` is called by the `ccipReceive` function, which ensures that only the router can deliver CCIP messages to the receiver contract.

Sending data and pay in LINK

The `sendMessagePayLINK` function undertakes five primary operations:

1. Call the `buildCCIPMessage` private function to construct a CCIP-compatible message using the `EVM2AnyMessage` struct:

- The receiver address is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through `abi.encode`.
- The data is encoded from a string to bytes using `abi.encode`.
- The `tokenAmounts` is an empty `EVMTokenAmount` struct array as no tokens are transferred.
- The `extraArgs` specifies the `gasLimit` for relaying the message to the recipient contract on the destination blockchain. In this example, the `gasLimit` is set to 200000.
- The `feeTokenAddress` designates the token address used for CCIP fees. Here, `address(linkToken)` signifies payment in LINK.

```
{" "}
```

```
<CcipCommon callout="extraArgsCallout" />
```

1. Computes the fees by invoking the router's `getFee` function.
1. Ensures your contract balance in LINK is enough to cover the fees.
1. Grants the router contract permission to deduct the fees from the contract's LINK balance.
1. Dispatches the CCIP message to the destination chain by executing the router's `ccipSend` function.

Note: As a security measure, the `sendMessagePayLINK` function is protected by the `onlyAllowlistedDestinationChain`, ensuring the contract owner has allowlisted a destination chain.

Sending data and pay in native

The `sendMessagePayNative` function undertakes four primary operations:

1. Call the `buildCCIPMessage` private function to construct a CCIP-compatible message using the `EVM2AnyMessage` struct:

- The receiver address is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through `abi.encode`.
- The data is encoded from a string to bytes using `abi.encode`.
- The `tokenAmounts` is an empty `EVMTokenAmount` struct array as no tokens are transferred.
- The `extraArgs` specifies the `gasLimit` for relaying the message to the recipient contract on the destination blockchain. In this example, the `gasLimit` is set to 200000.
- The `feeTokenAddress` designates the token address used for CCIP fees. Here, `address(0)` signifies payment in native gas tokens (ETH).

```
{" "}
```

```
<CcipCommon callout="extraArgsCallout" />
```

1. Computes the fees by invoking the router's getFee function.
1. Ensures your contract balance in native gas is enough to cover the fees.
1. Dispatches the CCIP message to the destination chain by executing the router's ccipSend function. Note: msg.value is set because you pay in native gas.

Note: As a security measure, the sendMessagePayNative function is protected by the onlyAllowlistedDestinationChain, ensuring the contract owner has allowlisted a destination chain.

Receiving data

On the destination blockchain, the router invokes the ccipReceive function which expects an Any2EVMMessage struct that contains:

- The CCIP messageId.
- The sourceChainSelector.
- The sender address in bytes format. Given that the sender is known to be a contract deployed on an EVM-compatible blockchain, the address is decoded from bytes to an Ethereum address using the ABI specifications.
- The data, which is also in bytes format. Given a string is expected, the data is decoded from bytes to a string using the ABI specifications.

This example applies three important security measures:

- ccipReceive is called by the ccipReceive function, which ensures that only the router can deliver CCIP messages to the receiver contract. See the onlyRouter modifier for more information.
- The modifier onlyAllowlisted ensures that only a call from an allowlisted source chain and sender is accepted.

```
# test-ccip-locally.mdx:
```

```
---
section: ccip
date: Last Modified
title: "Test CCIP Locally"
---
```

Chainlink Local provides a comprehensive set of tools and libraries to test your smart contracts with CCIP locally. By using Chainlink Local, you can quickly set up a local testing environment, simulate Chainlink services, and debug your contracts before deploying them to actual testnets.

Why Use Chainlink Local?

Testing your smart contracts locally can save you a significant amount of time and effort. With Chainlink Local, you can:

- Quickly identify and fix issues: Debug your contracts in a controlled environment before deploying them to testnets.
- Save time and resources: Reduce the need for repeated deployments to testnets, speeding up the development process.

Guides for Different Environments

Foundry

For Foundry users, the following guides will help you set up and test your CCIP smart contracts locally:

Foundry Guides for Chainlink Local

Hardhat

For Hardhat users, the following guides provide step-by-step instructions to integrate and test CCIP smart contracts locally:

Hardhat Guides for Chainlink Local

RemixIDE

For users who prefer RemixIDE, these guides will assist you in setting up and testing your CCIP smart contracts locally within the Remix environment:

RemixIDE Guides for Chainlink Local

usdc.mdx:

```
---
section: ccip
date: Last Modified
title: "Transfer USDC with Data"
---
```

```
import { Aside, ClickToZoom, CodeSample, CopyText } from "@components"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

USDC is a digital dollar backed 100% and is always redeemable 1:1 for US dollars. The stablecoin is issued by Circle on multiple blockchain platforms.

This guide will first explain how Chainlink CCIP enables native USDC transfers under the hood by leveraging Circle's Cross-Chain Transfer Protocol (CCTP). Then, you will learn how to use Chainlink CCIP to transfer USDC and arbitrary data from a smart contract on Avalanche Fuji to a smart contract on Ethereum Sepolia.

Note: In addition to programmable token transfers, you can also use CCIP to transfer USDC tokens without data. Check the Mainnets and Testnets configuration pages to learn on which blockchains CCIP supports USDC transfers.

Architecture

Fundamentally the architecture of CCIP and API are unchanged:

- The sender has to interact with the CCIP router to initiate a cross-chain transaction, similar to the process for any other token transfers. See the Transfer Tokens guide to learn more.
- The process uses the same onchain components including the Router, OnRamp, Commit Store, OffRamp, and Token Pool.
- The process uses the same offchain components including the Committing DON, Executing DON, and the Risk Management Network.
- USDC transfers also benefit from CCIP additional security provided by the Risk Management Network.

The diagram below shows that the USDC token pools and Executing DON handle the integration with Circle's contracts and offchain CCTP Attestation API. As with any other supported ERC-20 token, USDC has a linked token pool on each supported blockchain to facilitate OnRamp and OffRamp operations. To learn more about these components, read the architecture page.

<ClickToZoom src="/images/ccip/usdc-diagram.png" alt="Chainlink CCIP Detailed Architecture for usdc" />

The following describes the operational process:

1. On the source blockchain:
 1. When the sender initiates a transfer of USDC, the USDC token pool interacts with CCTP's contract to burn USDC tokens and specifies the USDC token pool address on the destination blockchain as the authorized caller to mint them.
 1. CCTP burns the specified USDC tokens and emits an associated CCTP event.
1. Offchain:
 1. Circle attestation service listens to CCTP events on the source blockchain.
 1. CCIP Executing DON listens to relevant CCTP events on the source blockchain. When it captures such an event, it calls the Circle Attestation service API to request an attestation. An attestation is a signed authorization to mint the specified amount of USDC on the destination blockchain.
1. On the destination blockchain:
 1. The Executing DON provides the attestation to the OffRamp contract.
 1. The OffRamp contract calls the USDC token pool with the USDC amount to be minted, the Receiver address, and the Circle attestation.
 1. The USDC token pool calls the CCTP contract. The CCTP contract verifies the attestation signature before minting the specified USDC amount into the Receiver.
 1. If there is data in the CCIP message and the Receiver is not an EOA, then the OffRamp contract transmits the CCIP message via the Router contract to the Receiver.

Example

In this tutorial, you will learn how to send USDC tokens from a smart contract on Avalanche Fuji to a smart contract on Ethereum Sepolia using Chainlink CCIP and pay CCIP fees in LINK tokens. The process uses the following steps:

1. Transfer USDC and Data: Initiate a transfer of USDC tokens and associated data from the Sender contract on Avalanche Fuji. The data includes the required arguments and the signature of the stake function from the Staker contract.
1. Receive and Stake: The Receiver contract on Ethereum Sepolia receives the tokens and data. Then, it uses this data to make a low-level call to the Staker contract, executing the stake function to stake USDC on behalf of a beneficiary.
1. Redeem Staked Tokens: The beneficiary can redeem the staked tokens for USDC later.

The purpose of including the function signature and arguments in the data is to demonstrate how arbitrary data can support a variety of scenarios and use cases. By sending specific instructions within the data, you can define various interactions between smart contracts across different blockchain networks and make your decentralized application more flexible and powerful.

<ClickToZoom src="/images/ccip/tutorials/usdc-tutorial.jpg" alt="Chainlink CCIP usdc tutorial" />

<Aside type="note">

After you followed the tutorial:

- You can read the Explanation section to understand the smart contracts' logic and how they interact with CCIP.
- Explore the CCIP USDC masterclass to find other examples of cross-chain transfers with USDC.

</Aside>

Before you begin

1. You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read this tutorial, which will guide you through using the Solidity programming language, interacting with the MetaMask wallet and working within the Remix Development Environment.
1. Your account must have some AVAX and LINK tokens on Avalanche Fuji and ETH tokens on Ethereum Sepolia. You can use the Chainlink faucet to acquire testnet tokens.
1. Check the Supported Networks page to confirm that USDC are supported for your lane. In this example, you will transfer tokens from Avalanche Fuji to Ethereum Sepolia so check the list of supported tokens here.
1. Use the Circle faucet to acquire USDC tokens on Avalanche Fuji.
1. Learn how to fund your contract. This guide shows how to fund your contract in LINK, but you can use the same guide for funding your contract with any ERC-20 tokens as long as they appear in the list of tokens in MetaMask.

Tutorial

Deploy your contracts

Deploy the Sender contract on Avalanche Fuji:

1. Open the Sender contract in Remix.

1. Compile your contract.

1. Deploy, fund your sender contract on Avalanche Fuji and enable sending messages to Ethereum Sepolia:

1. Open MetaMask and select the network Avalanche Fuji.

1. In Remix IDE, click on Deploy & Run Transactions and select Injected Provider - MetaMask from the environment list. Remix will then interact with your MetaMask wallet to communicate with Avalanche Fuji.

1. Fill in your blockchain's router, LINK, and USDC contract addresses. The router and USDC addresses can be found on the supported networks page and the LINK contract address on the LINK token contracts page. For Avalanche Fuji, the addresses are:

```

- Router address: <CopyText
text="0xf694e193200268f9a4868e4aa017a0118c9a8177" code/>
- LINK contract address: <CopyText
text="0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" code/>
- USDC contract address: <CopyText
text="0x5425890298aed601595a70AB815c96711a31Bc65" code/>

```

1. Click the transact button. After you confirm the transaction, the contract address appears on the Deployed Contracts list.

Note your contract address.

1. Open MetaMask and fund your contract with USDC tokens. You can transfer <CopyText text="1" code/> USDC to your contract.

1. Fund your contract with LINK tokens. You can transfer <CopyText text="1.5" code/> LINK to your contract. In this example, LINK is used to pay the CCIP fees.

Deploy the Staker and Receiver contracts on Ethereum Sepolia. Configure the Receiver contract to receive CCIP messages from the Sender contract:

1. Deploy the Staker contract:

1. Open MetaMask and select the network Ethereum Sepolia.

1. Open the Staker contract in Remix.

1. Compile your contract.

1. In Remix IDE, under Deploy & Run Transactions, make sure the environment is still Injected Provider - MetaMask.

1. Fill in the usdc contract address. The usdc contract address can be found


```

-----
----- |
| \destinationChainSelector | <CopyText text="16015286601757825753" code/>
<br /> The chain selector of Ethereum Sepolia. You can find it on the supported
networks page. |
| \receiver | Your receiver contract address at Ethereum
Sepolia. <br /> The receiver contract address.
|

```

1. Fill in the arguments of the setGasLimitForDestinationChain: function:

```

<br />

| Argument | Value and Description
|
| ----- |
-----
----- |
| \destinationChainSelector | <CopyText text="16015286601757825753" code/>
<br /> The chain selector of Ethereum Sepolia. You can find it on the supported
networks page. |
| \gasLimit | <CopyText text="200000" code/> <br /> The gas
limit for the execution of the CCIP message on the destination chain.
|

```

At this point:

- You have one sender contract on Avalanche Fuji, one staker contract and one receiver contract on Ethereum Sepolia.
- You enabled the sender contract to send messages to the receiver contract on Ethereum Sepolia.
- You set the gas limit for the execution of the CCIP message on Ethereum Sepolia.
- You enabled the receiver contract to receive messages from the sender contract on Avalanche Fuji.
- You funded the sender contract with USDC and LINK tokens on Avalanche Fuji.

Transfer and Receive tokens and data and pay in LINK

You will transfer 1 USDC and arbitrary data, which contains the encoded stake function name and parameters for calling Staker's stake function on the destination chain. The parameters contain the amount of staked tokens and the beneficiary address. The CCIP fees for using CCIP will be paid in LINK.

1. Transfer tokens and data from Avalanche Fuji:

1. Open MetaMask and select the network Avalanche Fuji.
1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Avalanche Fuji.
1. Fill in the arguments of the sendMessagePayLINK function:

```

<br />

| Argument | Value and Description
|
| ----- |
-----
----- |
| \destinationChainSelector | <CopyText text="16015286601757825753" code/>
<br /> CCIP Chain identifier of the destination blockchain (Ethereum Sepolia in
this example). You can find each chain selector on the supported networks page.
|

```

| \beneficiary | The beneficiary of the Staker tokens on Ethereum Sepolia. You can set your own EOA (Externally Owned Account) so you can redeem the Staker tokens in exchange for USDC tokens.

| \amount | <CopyText text="1000000" code/>
 The token amount (1 USDC).

1. Click on transact and confirm the transaction on MetaMask.
1. After the transaction is successful, record the transaction hash. Here is an example of a transaction on Avalanche Fuji.

<Aside type="note">

During gas price spikes, your transaction might fail, requiring more than 1.5 LINK to proceed. If your transaction fails, fund your contract with more LINK tokens and try again.

1. Open the CCIP explorer and search your cross-chain transaction using the transaction hash.

<ClickToZoom
src="/images/ccip/tutorials/ccip-explorer-send-usdc-message-pay-link-tx-details.jpg"
alt="Chainlink CCIP Explorer transaction details"
>

1. The CCIP transaction is completed once the status is marked as "Success". In this example, the CCIP message ID is 0xcb0fad9eec6664ad959f145cc4eb023924faded08baefc29952205ee37da7f13.

<ClickToZoom
src="/images/ccip/tutorials/ccip-explorer-send-usdc-message-pay-link-tx-details-success.jpg"
alt="Chainlink CCIP Explorer transaction details success"
>

1. Check the balance of the beneficiary on the destination chain:

1. Open MetaMask and select the network Ethereum Sepolia.
1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your Staker contract deployed on Ethereum Sepolia.
1. Call the balanceOf function with the beneficiary address.

<ClickToZoom
src="/images/ccip/tutorials/staker-tokens-balance.jpg"
alt="Chainlink CCIP Staker tokens balance"
>

1. Notice that the balance of the beneficiary is 1,000,000 Staker tokens. The Staker contract has the same number of decimals as the USDC token, which is 6. This means the beneficiary has 1 USDC staked and can redeem it by providing the same amount of Staker tokens.

1. Redeem the staked tokens:

1. Open MetaMask and make sure the network is Ethereum Sepolia.
1. Make sure you are connected with the beneficiary account.

1. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your Staker contract deployed on Ethereum Sepolia.

1. Call the redeem function with the amount of Staker tokens to redeem. In this example, the beneficiary will redeem 1,000,000 Staker tokens. When confirming, MetaMask will confirm that you will transfer the Staker tokens in exchange for USDC tokens.


```
<ClickToZoom
  src="/images/ccip/tutorials/staker-redeem-tokens.jpg"
  alt="Chainlink CCIP Avalanche message details"
/>
```

1. Confirm the transaction on MetaMask. After the transaction is successful, the beneficiary will receive 1 USDC tokens.


```
<ClickToZoom
  src="/images/ccip/tutorials/staker-redeem-tokens-success.jpg"
  alt="Chainlink CCIP Avalanche message details"
/>
```

Explanation

```
<CcipCommon callout="importCCIPPackage" />
```

The smart contracts featured in this tutorial are designed to interact with CCIP to send and receive USDC tokens and data across different blockchains. The contract code contains supporting comments clarifying the functions, events, and underlying logic. We will explain the Sender, Staker, and Receiver contracts further.

Sender Contract

```
<CodeSample src="samples/CCIP/usdc/Sender.sol" />
```

The Sender contract is responsible for initiating the transfer of USDC tokens and data. Hereâ€™s how it works:

1. Initializing the contract:

- When deploying the contract, you define the router address, LINK contract address, and USDC contract address.
- These addresses are essential for interacting with the CCIP router and handling token transfers.

1. sendMessagePayLINK function:

- This function sends USDC tokens, the encoded function signature of the stake function, and arguments (beneficiary address and amount) to the Receiver contract on the destination chain.
- Constructs a CCIP message using the EVM2AnyMessage struct.
- Computes the necessary fees using the routerâ€™s getFee function.
- Ensures the contract has enough LINK to cover the fees and approves the router transfer of LINK on its behalf.
- Dispatches the CCIP message to the destination chain by executing the routerâ€™s ccipSend function.
- Emits a MessageSent event.

Staker Contract

```
<CodeSample src="samples/CCIP/usdc/Staker.sol" />
```

The Staker contract manages the staking and redemption of USDC tokens. Hereâ€™s how it works:

1. Initializing the contract:

- When deploying the contract, you define the USDC token address.
- This address is essential for interacting with the USDC token contract.

1. stake function:

- Allows staking of USDC tokens on behalf of a beneficiary.
- Transfers USDC from the caller (msg.sender) to the contract, then mints an equivalent amount of staking tokens to the beneficiary.

1. redeem function:

- Allows beneficiaries to redeem their staked tokens for USDC.
- Burns the staked tokens and transfers the equivalent USDC to the beneficiary.

Receiver Contract

```
<CodeSample src="samples/CCIP/usdc/Receiver.sol" />
```

The Receiver contract handles incoming cross-chain messages, processes them, and interacts with the Staker contract to stake USDC on behalf of the beneficiary. Hereâ€™s how it works:

1. Initializing the Contract:

- When deploying the contract, you define the router address, USDC token address, and staker contract address.
- These addresses are essential for interacting with the CCIP router, USDC token, and Staker contracts.

1. ccipReceive function:

- The entry point for the CCIP router to deliver messages to the contract.
- Validates the sender and processes the message, ensuring it comes from the correct sender contract on the source chain.

1. Processing Message:

- Calls the processMessage function, which is external to leverage Solidityâ€™s try/catch error handling mechanism.
- Inside processMessage, it calls the ccipReceive function for further message processing.

1. ccipReceive function:

- Checks if the received token is USDC. If not, it reverts.
- Makes a low-level call to the stake function of the Staker contract using the encoded function signature and arguments from the received data.
- Emits a MessageReceived event upon successful processing.

1. Error Handling:

- If an error occurs during processing, the catch block within ccipReceive is executed.
- The messageId of the failed message is added to sfailedMessages, and the message content is stored in smessageContents.
- A MessageFailed event is emitted, allowing for later identification and reprocessing of failed messages.

1. retryFailedMessage function:

- Allows the contract owner to retry a failed message and recover the associated tokens.
- Updates the error code for the message to RESOLVED to prevent multiple retries.
- Transfers the locked tokens associated with the failed message to the specified beneficiary as an escape hatch.

1. getFailedMessages function:

- Retrieves a paginated list of failed messages for inspection.

index.mdx:

```
---
section: automation
date: Last Modified
title: "Chainlink Automation"
isIndex: true
whatsnext:
  {
    "Getting Started with Automation": "/chainlink-automation/overview/getting-
started",
    "Create Automation-Compatible Contracts":
"/chainlink-automation/guides/compatible-contracts",
    "Supported Networks for Automation":
"/chainlink-automation/overview/supported-networks",
    "Automation Architecture": "/chainlink-automation/concepts/automation-
architecture",
    "Automation Billing and Costs": "/chainlink-automation/overview/automation-
economics",
  }
---
```

```
import { Aside } from "@components"
import { ClickToZoom } from "@components"
import ChainlinkAutomation from
"@features/chainlink-automation/common/ChainlinkAutomation.astro"
```

```
<Aside type="note" title="Get Started">
  Try Chainlink Automation in the Getting Started guide.
</Aside>
```

```
<ChainlinkAutomation callout="deprecation" />
```

Automate your smart contracts using a secure and hyper-reliable decentralized network that uses the same external network of node operators that secures billions in value. Building on Chainlink Automation will accelerate your innovation, save you time and money, and help you get to market faster so you don't have to deal with the setup cost, ongoing maintenance, and risks associated with a centralized automation stack.

To learn more about how the Chainlink Automation Network automates your smart contracts, visit the Concepts and Architecture pages. You can also learn more through our additional Automation resources.

```
<ClickToZoom src="/images/automation/automation2diagramv3.png" />
```

Supported networks and costs

For a list of blockchains that are supported by Chainlink Automation, see the

Supported Networks page. To learn more about the cost of using Chainlink Automation, see the Automation Economics page.

Contact us

For help with your specific use case, contact us to connect with one of our Solutions Architects. You can also ask questions about Chainlink Automation on Stack Overflow or the #automation channel in our Discord server. For all developers resources, check out the Developer Resource Hub.

automation-architecture.mdx:

```
---
section: automation
date: Last Modified
title: "Automation Architecture"
isMdx: true
whatsnext:
  {
    "Automation Contracts": "/chainlink-automation/reference/automation-
contracts",
    "Automation Interfaces": "/chainlink-automation/reference/automation-
interfaces",
    "Automation Best Practices": "/chainlink-automation/concepts/best-practice",
  }
---
```

```
import { ClickToZoom } from "@components"
```

The following diagram describes the architecture of the Chainlink Automation Network. The Chainlink Automation Registry governs the actors on the network and compensates Automation Nodes for performing successful upkeeps. Developers can register their Upkeeps, and Node Operators can register as Automation Nodes.

```
<ClickToZoom src="/images/automation/automation2architecture.png" />
```

How it works

Automation Nodes form a peer-to-peer network using Chainlink's OCR3 protocol. Nodes use the Registry to know which Upkeeps to service and simulate checkUpkeep functions on their own block nodes to determine when upkeeps are eligible to be performed.

Using Chainlink's OCR3 protocol, nodes then get consensus on which upkeeps to perform and sign a report. The signed report contains the performData that will be executed onchain and the report is validated on the Registry before execution to provide cryptographic guarantees. The network has built-in redundancy and will still perform your upkeep even if some nodes are down.

```
<ClickToZoom src="/images/automation/automation2diagramv3.png" />
```

Chainlink Automation use the same battle-tested transaction manager mechanism built and used by Chainlink Data Feeds. This creates a hyper-reliable automation service that can execute and confirm transactions even during intense gas spikes or on chains with significant reorgs.

Internal monitoring

Internally, Chainlink Automation also uses its own monitoring and alerting mechanisms to maintain a healthy network and ensure developers get the industry leading reliability and performance.

Supported networks and costs

For a list of blockchains that is supported by Chainlink Automation, please review the supported networks page. To learn more about the cost of using Chainlink Automation, please review the Automation economics page.

```
# automation-concepts.mdx:
```

```
---
section: automation
date: Last Modified
title: "Automation Concepts"
isMdx: true
whatsnext:
  {
    "Automation Contracts": "/chainlink-automation/reference/automation-
contracts",
    "Automation Interfaces": "/chainlink-automation/reference/automation-
interfaces",
    "Automation Architecture": "/chainlink-automation/concepts/automation-
architecture",
  }
---
```

Before you explore how Chainlink Automation works on the architecture page, you should explore core concepts.

Prerequisites:

- Smart contracts
- ERC-677 Token Standard

Upkeeps and triggers

These are the jobs or tasks that you execute onchain. For example, you can call a smart contract function if a specific set of conditions are met. These specific conditions are called triggers. There are currently three types of triggers that the Chainlink Automation Network supports including:

- Time-based trigger: Use a time-based trigger to execute your function according to a time schedule. This feature is also called the Job Scheduler and it is a throwback to the Ethereum Alarm Clock. Time-based trigger contracts do not need to be compatible with the AutomationCompatibleInterface contract.
- Custom logic trigger: Use a custom logic trigger to provide custom solidity logic that Automation Nodes evaluate (offchain) to determine when to execute your function onchain. Your contract must meet the requirements to be compatible with the AutomationCompatibleInterface contract. Custom logic examples include checking the balance on a contract, only executing limit orders when their levels are met, any one of our coded examples, and many more.
- Log trigger: Use log data as both trigger and input. Your contract must meet the requirements to be compatible with the AutomationCompatibleInterface contract.

Automation nodes

Automation Nodes in the Chainlink Automation Network provide service to upkeeps that are funded and registered in the Automation registry. Automation Nodes use the same Node Operators as Chainlink Data Feeds.

Maximum logs processed for log trigger upkeeps

Chainlink Automation nodes look back over a limited range of the latest blocks on any particular chain. During this process, the nodes process a limited number of logs per block per upkeep, using a minimum dequeue method to ensure that the

latest logs are processed first. After this, the nodes may process additional remaining logs on a best effort basis, but this is not guaranteed. If you need all the remaining logs to be processed, configure a manual trigger as backup.

Expect the following numbers of logs to be processed:

Chain	Logs per block per upkeep
-----	-----
Ethereum	20
BSC (BNB)	4
Polygon	4
Avalanche	4
Gnosis	1
Optimism	4
BASE	4
Arbitrum	1 log every 2 blocks, or 2 logs per second

Note: Log triggers are not supported on Fantom.

```
# best-practice.mdx:
```

```
---
section: automation
date: Last Modified
title: "Automation Best Practices"
isMdx: true
whatsnext:
  {
    "Build Flexible Smart Contracts Using Automation":
"/chainlink-automation/guides/flexible-upkeeps",
    "Manage your Upkeeps": "/chainlink-automation/guides/manage-upkeeps",
  }
---
```

```
import { Aside } from "@components"
```

This guide outlines the best practices when using Chainlink Automation. These best practices are important for using Chainlink Automation securely and reliably when you create Automation-compatible contracts.

Use the latest version of Chainlink Automation

To get the best reliability and security guarantees for your upkeep, use the latest version of Chainlink Automation.

```
<Aside type="caution" title="Deprecation of older upkeeps">
  Migrate existing upkeeps before August 29, 2024. Versions earlier
  than 2.1 are no longer supported, and existing upkeeps on versions earlier
  than 2.1 will stop being performed on
  August 29, 2024.
</Aside>
```

Make registry and registrar addresses configurable

Where your upkeep calls the registry or the registrar, you must make the address configurable so you can migrate your upkeep easily with the one-click migration capability. If you don't make the address configurable, you must redeploy the upkeep for migrations.

Alternatively, set the forwarder address when your upkeep is deployed and read the registry from the forwarder during your execution to simplify it.

Use the Forwarder

If your upkeep performs sensitive functions in your protocol, consider using the Forwarder to lock it down so performUpkeep can only be called by the Forwarder. Add other permissible addresses if you need to call it yourself. Note the forwarder is only determined after registration so make this a mutable variable and ensure you add a setter function with permissions for you to set it.

Verify Data Streams reports fetched with StreamsLookup

If your upkeep uses StreamsLookup, ensure you use the verification interface to verify your reports onchain.

Avoid "flickering" custom logic upkeeps

The Automation DON evaluates your upkeep regularly. When your upkeep is eligible, the DON attempts to perform the upkeep. For best results, ensure that checkUpkeep remains true until execution.

If the state of your upkeep "flickers" by rapidly alternating between true and false, then your upkeep is at risk of not being performed. "Flickering" upkeeps might cause unintended consequences because there is latency between observing the state of the chain, getting consensus (v2 and later) on what needs to happen, and confirming the transaction onchain.

Always test your contracts

As with all smart contract testing, it is important to test the boundaries of your smart contract in order to ensure it operates as intended. Similarly, it is important to make sure the compatible contract operates within the parameters of the Registry.

Test all of your mission-critical contracts, and stress-test the contract to confirm the performance and correct operation of your use case under load and adversarial conditions. The Chainlink Automation Network will continue to operate under stress, but so should your contract. For a list of supported testnet blockchains, please review the supported networks page.

Using ERC-677 tokens

For registration on Mainnet, you need ERC-677 LINK. Many token bridges give you ERC-20 LINK tokens. Use PegSwap to convert Chainlink tokens (LINK) to be ERC-677 compatible. To register on a supported testnet, get LINK for the testnet that you want to use from our faucet.

Automation v1 upkeep revalidation

If your upkeep is on Automation v1, we recommend that you revalidate the conditions specified in checkUpkeep in your performUpkeep function. Automation v1 uses a turn taking system where nodes rotate to monitor your upkeep. It does not use consensus.

compatible-contracts.mdx:

section: automation

date: Last Modified

title: "Create Automation-Compatible Contracts"

isMdx: true

whatsnext:

{

 "Access Data Streams Using Automation":

 "/chainlink-automation/guides/streams-lookup",

 "Register Custom Logic Upkeeps": "/chainlink-automation/guides/register-

```
upkeep",
  "Build Flexible Smart Contracts Using Automation":
    "/chainlink-automation/guides/flexible-upkeeps",
  "Manage your Upkeeps": "/chainlink-automation/guides/manage-upkeeps",
}
---
```

```
import { Aside, CodeSample } from "@components"
```

Learn how to make smart contracts that are compatible with Automation.

```
<Aside type="tip" title="Considerations and Best Practices">
```

Before you deploy contracts to use with Chainlink Automation, read the Best Practices guide. These best practices are important for using Chainlink Automation securely and reliably. You can also read more about the Chainlink Automation architecture here.

```
</Aside>
```

Automation compatible contracts

A contract is Automation-compatible when it follows a specified interface that allows the Chainlink Automation Network to determine if, when, and how the contract should be automated.

The interface you use will depend on the type of trigger you want to use:

- If you want a log event to trigger your upkeep, use the `ILogAutomation` interface.
- If you want to use onchain state in a custom calculation to trigger your upkeep, use `AutomationCompatibleInterface` interface.
- If you want to call a function just based on time, you don't need an interface. Consider instead using a time-based upkeep.
- If you want to use Automation with Data Streams, use `StreamsLookupCompatibleInterface` interface.

You can learn more about these interfaces here.

Example Automation-compatible contract using custom logic trigger

Custom logic Automation compatible contracts must meet the following requirements:

- Import `AutomationCompatible.sol`. You can refer to the Chainlink Contracts on GitHub to find the latest version.
- Use the `AutomationCompatibleInterface` from the library to ensure your `checkUpkeep` and `performUpkeep` function definitions match the definitions expected by the Chainlink Automation Network.
- Include a `checkUpkeep` function that contains the logic that will be executed offchain to see if `performUpkeep` should be executed. `checkUpkeep` can use onchain data and a specified `checkData` parameter to perform complex calculations offchain and then send the result to `performUpkeep` as `performData`.
- Include a `performUpkeep` function that will be executed onchain when `checkUpkeep` returns true.

Use these elements to create a compatible contract that will automatically increment a counter after every `updateInterval` seconds. After you register the contract as an upkeep, the Chainlink Automation Network frequently simulates your `checkUpkeep` offchain to determine if the `updateInterval` time has passed since the last increment (timestamp). When `checkUpkeep` returns true, the Chainlink Automation Network calls `performUpkeep` onchain and increments the counter. This cycle repeats until the upkeep is cancelled or runs out of funding.

```
<CodeSample src="samples/Automation/AutomationCounter.sol" />
```

Compile and deploy your own Automation Counter onto a supported Testnet.

1. In the Remix example, select the compile tab on the left and press the compile button. Make sure that your contract compiles without any errors. Note that the Warning messages in this example are acceptable and will not block the deployment.

1. Select the Deploy tab and deploy the Counter smart contract in the injected web3 environment. When deploying the contract, specify the `updateInterval` value. For this example, set a short interval of 60. This is the interval at which the `performUpkeep` function will be called.

1. After deployment is complete, copy the address of the deployed contract. This address is required to register your upkeep in the Automation UI. The example in this document uses custom logic automation.

To see more complex examples, go to the Quick Starts page.

Now register your upkeep.

Vyper example

```
<Aside type="note" title="Note on arrays">
```

Make sure the `checkdata` array size is correct. Vyper does not support dynamic arrays.

```
</Aside>
```

You can find a `KeepersConsumer` example [here](#). Read the `apeworx-starter-kit` README to learn how to run the example.

```
# flexible-upkeeps.mdx:
```

```
---
section: automation
date: Last Modified
title: "Create Flexible, Secure, and Low-Cost Smart Contracts"
isMdx: true
whatsnext:
  {
    "Manage your Upkeeps": "/chainlink-automation/guides/manage-upkeeps",
    "Troubleshoot and Debug Upkeeps":
"/chainlink-automation/reference/debugging-errors",
  }
---
```

```
import { Aside, CodeSample } from "@components"
```

In this guide, you will learn how the flexibility of Chainlink Automation enables important design patterns that reduce gas fees, enhance the resilience of dApps, and improve end-user experience. Smart contracts themselves cannot self-trigger their functions at arbitrary times or under arbitrary conditions. Transactions can only be initiated by another account.

Start by integrating an example contract to Chainlink Automation that has not yet been optimized. Then, deploy a comparison contract that shows you how to properly use the flexibility of Chainlink Automation to perform complex computations without paying high gas fees.

Prerequisites

This guide assumes you have a basic understanding of Chainlink Automation. If you are new to Automation, complete the following guides first:

- Learn how to deploy solidity contracts using Remix and Metamask
- Learn how to make compatible contracts
- Register Upkeep for a smart contract

Chainlink Automation is supported on several networks.

<Aside type="note" title="ERC677 Link">

- Get LINK on the supported testnet that you want to use.
- For funding on Mainnet, you need ERC-677 LINK. Many token bridges give you ERC-20 LINK tokens. Use PegSwap to convert Chainlink tokens (LINK) to be ERC-677 compatible.

</Aside>

Problem: Onchain computation leads to high gas fees

In the guide for Creating Compatible Contracts, you deployed a basic counter contract and verified that the counter increments every 30 seconds. However, more complex use cases can require looping over arrays or performing expensive computation. This leads to expensive gas fees and can increase the premium that end-users have to pay to use your dApp. To illustrate this, deploy an example contract that maintains internal balances.

The contract has the following components:

- A fixed-size(1000) array balances with each element of the array starting with a balance of 1000.
- The withdraw() function decreases the balance of one or more indexes in the balances array. Use this to simulate changes to the balance of each element in the array.
- Automation Nodes are responsible for regularly re-balancing the elements using two functions:
 - The checkUpkeep() function checks if the contract requires work to be done. If one array element has a balance of less than LIMIT, the function returns upkeepNeeded == true.
 - The performUpkeep() function to re-balances the elements. To demonstrate how this computation can cause high gas fees, this example does all of the computation within the transaction. The function finds all of the elements that are less than LIMIT, decreases the contract liquidity, and increases every found element to equal LIMIT.

<CodeSample src="samples/Automation/BalancerOnChain.sol" />

Test this example using the following steps:

1. Deploy the contract using Remix on the supported testnet of your choice.

1. Before registering the upkeep for your contract, decrease the balances of some elements. This simulates a situation where upkeep is required. In Remix, Withdraw 100 at indexes 10,100,300,350,500,600,670,700,900. Pass 100, [10,100,300,350,500,600,670,700,900] to the withdraw function:

```
!Withdraw 100 at 10,100,300,350,500,600,670,700,900
```

You can also perform this step after registering the upkeep if you need to.

1. Register the upkeep for your contract as explained here. Because this example has high gas requirements, specify the maximum allowed gas limit of 2,500,000.

1. After the registration is confirmed, Automation Nodes perform the upkeep.

```
!BalancerOnChain Upkeep History
```

1. Click the transaction hash to see the transaction details in Etherscan. You can find how much gas was used in the upkeep transaction.

!BalancerOnChain Gas

In this example, the `performUpkeep()` function used 2,481,379 gas. This example has two main issues:

- All computation is done in `performUpkeep()`. This is a state modifying function which leads to high gas consumption.
- This example is simple, but looping over large arrays with state updates can cause the transaction to hit the gas limit of the network, which prevents `performUpkeep` from running successfully.

To reduce these gas fees and avoid running out of gas, you can make some simple changes to the contract.

Solution: Perform complex computations with no gas fees

Modify the contract and move the computation to the `checkUpkeep()` function. This computation doesn't consume any gas and supports multiple upkeepes for the same contract to do the work in parallel. The main difference between this new contract and the previous contract are:

- The `checkUpkeep()` function receives `checkData`, which passes arbitrary bytes to the function. Pass a `lowerBound` and an `upperBound` to scope the work to a sub-array of balances. This creates several upkeeps with different values of `checkData`. The function loops over the sub-array and looks for the indexes of the elements that require re-balancing and calculates the required increments. Then, it returns `upkeepNeeded == true` and `performData`, which is calculated by encoding indexes and increments. Note that `checkUpkeep()` is a view function, so computation does not consume any gas.
- The `performUpkeep()` function takes `performData` as a parameter and decodes it to fetch the indexes and the increments.

```
<CodeSample src="samples/Automation/BalancerOffChain.sol" />
```

Run this example to compare the gas fees:

1. Deploy the contract using Remix on the supported testnet of your choice.

1. Withdraw 100 at 10,100,300,350,500,600,670,700,900. Pass 100, [10,100,300,350,500,600,670,700,900] to the withdraw function the same way that you did for the previous example.

1. Register three upkeeps for your contract as explained here. Because the Automation Nodes handle much of the computation offchain, a gas limit of 200,000 is sufficient. For each registration, pass the following checkData values to specify which balance indexes the registration will monitor. Note: You must remove any breaking line when copying the values.

```
| Upkeep Name | CheckData(base16)
| Remark: calculated using abi.encode() |
| ----- |
-----
-----
-----
-----
-----
| balancerOffChainSubset1 |
0x0000000000000000000000000000000000000000000000000000000000000000<br/>0000000000000000000000000000000000000000000000000000000000000000<br/>0000000000000000000000000000000000000000000000000000000000000000<br/>0000000000000000000000000000000000000000000000000000000000000014c |
lowerBound: 0<br/>upperBound: 332
|
```


Each registered upkeep under the Chainlink Automation network has its own unique Forwarder contract. The Forwarder address will only be known after registration, as we deploy a new forwarder for each upkeep. The Forwarder contract is the intermediary between the Automation Registry and your Upkeep contract. The Forwarder is always the msg.Sender for your upkeep.

If your performUpkeep function is open and callable by anyone without risk of accepting unintentional external data, you don't need to use the Forwarder.

Securing your upkeep

If your upkeep's perform function needs to be permissioned, please consider adding msg.sender = forwarder at the top of your performUpkeep function.

To make this work you will need to:

- Create forwarder as a mutable address variable on your contract that only you can update. forwarder is a unique value that cannot change for your upkeep.
- Create setForwarder function so you can update the forwarder address.
- After registration run setForwarder with the forwarder address in your UI, or programmatically fetch it using registry.getForwarder(upkeepID) using the Registry interface.

Code example

The code sample below uses the Forwarder:

```
<CodeSample src="samples/Automation/CounterwForwarder.sol" />
```

```
# gas-price-threshold.mdx:
```

```
---
section: automation
date: Last Modified
title: "Set a gas price threshold on your upkeep"
isMdx: true
whatsnext:
  {
    "Register Custom Logic Upkeeps": "/chainlink-automation/guides/register-
upkeep",
    "Register Log Trigger Upkeeps": "/chainlink-automation/guides/log-trigger",
    "Automation Architecture": "/chainlink-automation/concepts/automation-
architecture",
    "Billing and Costs": "/chainlink-automation/overview/automation-economics",
  }
---

import { Aside, ClickToZoom } from "@components"
import { Tabs } from "@components/Tabs"
```

You can set a gas price threshold for your upkeep to prevent it from being serviced when gas prices exceed the maximum gas price you specify. You can set a maximum gas price on conditional upkeeps and log trigger upkeeps.

Note: This is a different gas setting than the upkeep gas limit, which limits the amount of gas used.

After you set your maximum gas price, it takes a few minutes to go into effect as it syncs with the nodes. Updating your maximum gas price does not impact any upkeep where the execution is in-flight.

Limitations

Do not set a gas price threshold when speed of execution is important. The gas

price threshold can significantly delay the execution of conditional upkeeps and prevent execution entirely for log trigger upkeeps.

Gas prices spike after your transaction is in flight

Due to the decentralized nature of the Automation network and its transaction manager, your upkeep may be performed at a gas price higher than the maximum gas price you set. This edge case can happen when:

1. The Automation network determines that the upkeep should be performed while the onchain gas price is below your threshold. After that check occurs, the performUpkeep transaction is in flight (in the mempool).
1. After the transaction is already in flight, gas prices start spiking and move above your threshold. To ensure that the node's sending key nonce does not become blocked, the node automatically increases the gas to confirm the transaction and prevent the nonce from blocking subsequent transactions.

To avoid this edge case, increase the buffer in your gas price threshold.

Log trigger upkeeps are not retried

If a log trigger upkeep is triggered by a log event, and is declared ineligible to perform the upkeep because gas prices are above your gas price threshold, the upkeep is not retried.

Choose your maximum gas price

Each node compares the specified threshold to its estimate of onchain gas price. A quorum is needed before upkeeps are performed. Nodes may bid aggressively on gas prices to ensure transactions go through. If the node's gas price bid is above your maximum gas price, the node will not perform your upkeep. For example, if you set a gas price threshold of 3 gwei, your upkeep's execution may stop as soon as the node's gas price bid hits 3 gwei, even if the actual gas price is only 2 gwei. To adjust for this, you may need to set a higher gas price threshold.

Set the maximum gas price on an existing upkeep

Set the maximum gas price using the offchainConfig field in your upkeep. Only the upkeep admin can set gas controls for an upkeep. This setting is not yet available in the Chainlink Automation App, so it must be done programmatically.

To set the maximum gas price on an upkeep, follow these steps:

1. Format and encode your offchain config. The offchain config is where you set your maximum gas price.
1. Run setUpkeepOffchainConfig on the registry using your upkeep ID and the encoded value of your offchain config.

Run the script

The Automation gas threshold script encodes and sets your offchain config, which includes the maximum gas price in wei.

1. To run this example, clone the repo and install its dependencies:

```
sh
git clone https://github.com/smartcontractkit/smart-contract-examples.git &&
cd smart-contract-examples/automation-gas-threshold
```

```
sh
npm install
```


- YOURRPCURL: The RPC URL for your provider (such as Alchemy or Infura)
- YOURPRIVATEKEY: Your wallet's private key
- YOURUPKEEPID: The ID of the Automation upkeep you want to configure.
- Within the offchainConfig variable, set your maxGasPrice in wei. Do not use quotation marks around the value you set for maxGasPrice. If this string is formatted incorrectly, the feature does not work. Here's an example of correct formatting: {"maxGasPrice":20000000000}

```
sh
node index.js
```

Remove the maximum gas price

To remove the maximum gas price from your upkeep, set the value of your `offchainConfig` back to `0x00`:

- Encode this request with CBOR encoding.
- Run `setUpkeepOffchainConfig` on the registry using your upkeep ID and the CBOR encoded value for `0x00`.

If you're using the gas threshold script, set the `offchainConfig` variable in the script to 0:

```
js
// Change this value from {"maxGasPrice":20000000000} to 0:
const offchainConfig = 0
```

Create a new upkeep with a gas price threshold

To create a new upkeep with a gas threshold in place, you can create a conditional upkeep or log trigger upkeep programmatically. Note: The Chainlink Automation App does not yet support setting a gas price threshold when creating a new upkeep.

You need to format and encode your offchain config before you set the `offchainConfig` parameter. You can adjust the gas threshold script to get the encoded value for your offchain config and set that as the `offchainConfig` variable when creating your new upkeep, or you can encode your config using the Solidity or Go examples below.

Format and encode your offchain config

Currently, the only parameter that you can set in your upkeep's offchain config is `maxGasPrice`. You need to format your offchain config as a JSON object and CBOR encode it before you update it on the registry.

1. Format your offchain config as JSON. For example: `{"maxGasPrice": 10000000000000}`.
Use quotation marks only around the key, "maxGasPrice". Do not use quotation marks around the value of the maximum gas price you are setting.
1. Encode the JSON object using CBOR encoding:

```
{/ prettier-ignore /}  
<Tabs sharedStore="scLang" client:visible>  
<Fragment slot="tab.1">Solidity</Fragment>  
<Fragment slot="tab.2">Go</Fragment>
```

```

<Fragment slot="panel.1">
solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import { CBOR } from "@chainlink/contracts/src/v0.8/vendor/solidity-
cborutils/v2.0.0/CBOR.sol"

contract CBOREncoder {
using CBOR for CBOR.CBORBuffer;

// @notice encodes a max gas price to CBOR encoded bytes
// @param maxGasPrice The max gas price
// @return CBOR encoded bytes and the struct depth
function encode(uint256 maxGasPrice, uint256 capacity) external pure
returns (bytes memory, uint256) {
CBOR.CBORBuffer memory buffer = CBOR.create(capacity);
buffer.writeString("maxGasPrice");
buffer.writeUInt256(maxGasPrice);
return (buffer.buf.buf, buffer.depth);
}
}

</Fragment>
<Fragment slot="panel.2">
For Go, create a simple struct like this and encode it with the cbor
package.
go
type UpkeepOffchainConfig struct {
    MaxGasPrice big.Int json:"maxGasPrice" cbor:"maxGasPrice"
}

</Fragment>
</Tabs>

```

job-scheduler.mdx:

```

---
section: automation
date: Last Modified
title: "Create a Time-Based Upkeep"
isMdx: true
whatsnext:
{
    "Register Custom Logic Upkeeps": "/chainlink-automation/guides/register-
upkeep",
    "Register Log Trigger Upkeeps": "/chainlink-automation/guides/log-trigger",
    "Automation Architecture": "/chainlink-automation/concepts/automation-
architecture",
    "Billing and Costs": "/chainlink-automation/overview/automation-economics",
}
---

```

```
import { Aside, ClickToZoom } from "@components"
```

Create powerful automation for your smart contract using time schedules without having to create Automation-compatible contracts. This guide explains how to register time-based upkeeps.

!Job Scheduler animation

```

<Aside type="tip" title="Testing and best practices">
    Follow the best practices when creating an Automation-compatible

```

contract and test your upkeep on a testnet before deploying it to a mainnet.
</Aside>
Using the Chainlink Automation app

In the Chainlink Automation App, click the blue Register new Upkeep button.

<ClickToZoom src="/images/automation/auto-ui-home.png" />

Connecting your wallet

If you do not already have a wallet connected with the Chainlink Automation network, the interface will prompt you to do so. Click the Connect Wallet button and follow the remaining prompts to connect your wallet to one of the Automation supported blockchain networks.

<ClickToZoom src="/images/automation/auto-ui-wallet.png" />

Trigger selection

Select Time-based trigger.

<ClickToZoom src="/images/automation/uiselecttrigger.png" />

Using time-based triggers

When you select the time-based trigger, you are prompted to enter a contract address. Provide the address of the contract you want to automate. If you did not verify the contract on chain, you will need to paste the Application Binary Interface (ABI) of the deployed contract into the corresponding text box. Select the function name that you want to execute and provide any static inputs. If you want to use dynamic inputs please see Custom logic Upkeeps

<ClickToZoom src="/images/automation/automation-time-based-trigger.png" />

Specifying the time schedule

After you have successfully entered your contract address and ABI, specify your time schedule in the form of a CRON expression. CRON expressions provide a shorthand way of creating a time schedule. You can use the provided example buttons in the Automation app to experiment with different schedules. Then, create your own time schedule.

Cron jobs are interpreted according to this format:

â"€â" â" â" â" â" â" â" â" â" â" â" â" minute (0 - 59)
â", â"â" â" â" â" â" â" â" â" â" â" â" â" hour (0 - 23)
â", â", â"â" â" â" â" â" â" â" â" â" â" day of the month (1 - 31)
â", â", â", â"â" â" â" â" â" â" â" â" â" â" month (1 - 12)
â", â", â", â", â", â"â" â" â" â" â" â" â" â" â" â" day of the week (0 - 6)
(Sunday to Saturday)
â", â", â", â", â",
â", â", â", â", â",
â", â", â", â", â",

All times are in UTC.

- can be used for range e.g. "0 8-16 "
/ can be used for interval e.g. "0 /2 "
, can be used for list e.g. "0 17 0,2,4"

Special limitations:
there is no year field

no special characters: ? L W #
lists can have a max length of 26
no words like JAN / FEB or MON / TUES

After entering your CRON expression, click Next.

<ClickToZoom src="/images/automation/automation-cron-expression.png" />

Entering upkeep details

Provide the following information in the Automation app:

- Upkeep name: This will be visible in the Chainlink Automation app.
- Gas limit: This is the maximum amount of gas that your transaction requires to execute on chain. This limit cannot exceed the performGasLimit value configured on the registry.

<Aside type="note" title="Job Scheduler Gas requirements">

When you create an upkeep through the Job Scheduler, Chainlink Automation deploys a new CronUpkeep contract from the CronUpkeepFactory to manage your time schedule and ensure that it is compatible. Because this contract uses roughly 110K gas per call, it is recommended to add 150K additional gas to the gas limit of the function you would like to automate.

</Aside>

- Starting balance (LINK): Specify a LINK starting balance to fund your upkeep. See the LINK Token Contracts page to find the correct contract address and access faucets for testnet LINK. This field is required. You must have LINK before you can use the Chainlink Automation service.

<Aside type="tip" title="ERC-677 Link">

For registration you must use ERC-677 LINK. Read our LINK page to determine where to acquire mainnet LINK, or visit faucets.chain.link to request testnet LINK.

</Aside>

- Your email address (optional): This email address will be used to send you an email notification when your upkeep is underfunded.

<ClickToZoom src="/images/automation/automation-upkeep-details.png" />

Complete upkeep registration

Click Register upkeep and confirm the transaction in MetaMask.
!Upkeep Registration Success Message

Your upkeeps will be displayed in your list of Active Upkeeps. You must monitor the balance of your upkeep. If the balance drops below the minimum balance, the Chainlink Automation Network will not perform the Upkeep. See Managing Upkeeps to learn how to manage your upkeeps.

log-trigger.mdx:

section: automation
date: Last Modified
title: "Register a Log Trigger Upkeep"

```
isMdx: true
whatsnext:
  {
    "Automation Interfaces": "/chainlink-automation/reference/automation-interfaces",
    "Register Upkeeps Programmatically": "/chainlink-automation/guides/register-upkeep-in-contract",
    "Automation Architecture": "/chainlink-automation/concepts/automation-architecture",
  }
---
```

```
import { Aside, CodeSample, ClickToZoom } from "@components"
```

Create powerful smart contracts that use log data as both trigger and input. This guide explains how to create log-trigger upkeeps.

```
<Aside type="tip" title="Testing and best practices">
  Follow the best practices when creating a compatible contract and test
  your upkeep on a testnet before deploying it to a mainnet.
</Aside>
```

Understanding maximum logs processed

Chainlink Automation processes a limited number of logs per block per upkeep. See the Concepts page to learn about how logs are processed and how many logs you can expect to be processed per block on the chain you're using.

Emit a log

1. Open CountEmitLog.sol in Remix. This contract contains an event WantsToCount that keeps track of the address of the message sender. The function emitCountLog emits this event.

```
{" "}
```

```
<CodeSample src="samples/Automation/CountEmitLog.sol" />
```

1. Under Environment, select the option Injected Provider to connect to your cryptocurrency wallet.

1. Deploy the contract and confirm the transaction.

1. You can view the contract on Etherscan by clicking the message in the terminal. You can view the address of the created contract and the original contract in Etherscan.

```
<ClickToZoom src="/images/automation/log-trig-addresses.png" />
```

1. Navigate to the Contract tab. If the contract is already verified, you will see options to Read Contract and Write Contract. If your contract isn't verified, follow the prompts in Etherscan to verify the contract.

1. Click Write Contract and click the emitCountLog button to emit a log.

1. Navigate back to Etherscan and locate the Events tab. You should see the event of emitting a log recorded in this section.

```
<ClickToZoom src="/images/automation/log-trig-event.png" />
```

Using ILogAutomation Interface

1. Open CountWithLog.sol in Remix. This contract contains a struct to account for the log structure and uses the ILogAutomation interface for log automation. The interface contains the checkLog and performUpkeep functions. The contract contains an event CountedBy. The counted variable will be incremented when performUpkeep is called.

```
{" "}
```

```
<CodeSample src="samples/Automation/CountWithLog.sol" />
```

1. Deploy the contract and confirm your transaction.
1. Under Deployed Contracts, expand CountWithLog. Click the count button to view the value of the count variable. It should be 0.
<ClickToZoom src="/images/automation/log-trig-count-0.png" />
1. Copy the address of this contract either via Remix or Etherscan to register it on the Chainlink Automation app.

Using the Chainlink Automation App

```
<div class="remix-callout">  
  <a href="https://automation.chain.link">Open the Chainlink Automation App</a>  
</div>
```

Click the Register New Upkeep button.

```
<ClickToZoom src="/images/automation/auto-ui-home.png" />
```

Connecting your wallet

If you do not already have a wallet connected with the Chainlink Automation network, the interface will prompt you to do so. Click the Connect Wallet button and follow the remaining prompts to connect your wallet to one of the Automation supported blockchain networks.

```
<ClickToZoom src="/images/automation/auto-ui-wallet.png" />
```

Trigger selection

Select Log Trigger.

```
<ClickToZoom src="/images/automation/uiselecttrigger.png" />
```

Using log triggers

```
<Aside type="tip" title="Reorg protection">  
  Your upkeeps will be protected against logs that are emitted during a reorg.  
</Aside>
```

1. Provide the address of your Automation-compatible contract that you want to automate. In this case, we will paste the address of CountWithLog.sol. This contract must follow the format of the ILogAutomation interface to ensure Automation nodes can interact with your contract as expected.

```
<ClickToZoom src="/images/automation/logtrig1upkeepaddress.png" />
```

1. Provide the address of the contract that will be emitting the log. In this case, this is the address of CountEmitLog.sol. If the contract is not validated you will need to provide the ABI.

```
<ClickToZoom src="/images/automation/logtrig2emitteraddress.png" />
```

To find the ABI of your contract in Remix, navigate to the Compiler view using the left side icons. Then, copy the ABI to your clipboard using the button at the bottom of the panel.

```
<ClickToZoom src="/images/automation/log-trig-ebi.png" />
```

1. Use the dropdown to select the triggering event. This is WantsToCount. You can also provide one optional filter per any of the indexed events in the log, but you don't have to. When this combination of filters are matched the upkeep will trigger.

```
<ClickToZoom src="/images/automation/logtrig3logsigfilterpopulated.png" />
```

Entering upkeep details

Provide the following information in the Automation app:

- Upkeep name: This will be visible in the Chainlink Automation app.
- Gas limit: This is the maximum amount of gas that your transaction requires to execute on chain. This limit cannot exceed the performGasLimit value configured on the registry.
- Starting balance (LINK): Specify a LINK starting balance to fund your upkeep. See the LINK Token Contracts page to find the correct contract address and access faucets for testnet LINK. This field is required. You must have LINK before you can use the Chainlink Automation service.

```
{/ prettier-ignore /}
```

```
{" "}
```

```
<Aside type="tip" title="Funding Upkeep">
```

You should fund your contract with more LINK than you anticipate you will need. The network will not check or perform your Upkeep if your balance is too low based on current exchange rates. View the Automation economics page to learn more about the cost of using Chainlink Automation.

```
</Aside>
```

```
{/ prettier-ignore /}
```

```
{" "}
```

```
<Aside type="tip" title="ERC-677 Link">
```

For funding on Mainnet, you need ERC-677 LINK. Many token bridges give you ERC-20 LINK tokens. Use PegSwap to convert Chainlink tokens (LINK) to be ERC-677 compatible. Use faucets.chain.link to get testnet LINK.

```
</Aside>
```

- Check data: Optional input field that you may use depending on whether you are using it in your contract.

- Your email address (optional): This email address will be used to send you an email notification when your upkeep is underfunded.

Complete upkeep registration

Click Register upkeep and confirm the transaction in MetaMask.

!Upkeep Registration Success Message

Your upkeeps will be displayed in your list of Active Upkeeps. You must monitor the balance of your upkeep. If the balance drops below the minimum balance, the Chainlink Automation Network will not perform the Upkeep. See Managing Upkeeps to learn how to manage your upkeeps.

```
<ClickToZoom src="/images/automation/log-trig-config.png" />
```

Performing upkeep

Navigate back to the Etherscan page for CountEmitLog.sol. Under Write Contract, click the button to emitCountLog. Refresh the upkeep details page. You may have to wait a few moments. Under History, you should see the upkeep has been performed.

manage-upkeeps.mdx:

```

---
section: automation
date: Last Modified
title: "Managing Upkeeps"
isMdx: true
whatsnext:
  {
    "Troubleshoot and Debug Upkeeps":
      "/chainlink-automation/reference/debugging-errors",
    "Automation Billing and Costs": "/chainlink-automation/overview/automation-
economics",
  }
---

```

```

import { Aside } from "@components"
import ChainlinkAutomation from
"@features/chainlink-automation/common/ChainlinkAutomation.astro"

```

```
<ChainlinkAutomation callout="deprecation" />
```

Manage your Upkeeps to get the best performance.

Fund your upkeep

You must monitor the balance of your Upkeep. If the Upkeep LINK balance drops below the minimum balance, the Chainlink Automation Network will not perform the Upkeep.

```

<Aside type="tip" title="ERC-677 Link">
  For funding on Mainnet, you need ERC-677 LINK. Many token bridges give you
ERC-20 LINK tokens. Use PegSwap to convert
  Chainlink tokens (LINK) to be ERC-677 compatible. Use
  faucets.chain.link to get testnet LINK.
</Aside>

```

Follow these steps to fund your Upkeep:

1. Click View Upkeep or go to the Chainlink Automation App and click on your recently registered Upkeep under My Upkeeps.

1. Click the Add funds button

1. Approve the LINK spend allowance
!Approve LINK Spend Allowance

1. Confirm the LINK transfer by sending funds to the Chainlink Automation Network Registry
!Confirm LINK Transfer

1. Receive a success message and verify that the funds were added to the Upkeep
!Funds Added Successful Message

Maintain a minimum balance

Each Upkeep has a minimum balance to ensure that an Upkeeps will still run should a sudden spike occur. If your Upkeep LINK balance drops below this amount, the Upkeep will not be performed.

To account for Upkeep execution over time and possible extended gas spikes, maintain an Upkeep LINK balance that is 3 to 5 times the minimum balance. Note if you have an upkeep that performs frequently you may want to increase the buffer to ensure a reasonable interval before you need to fund again. Developers also have the ability to update performGasLimit for an upkeep.

Withdraw funds

To withdraw funds, the Upkeep administrator have to cancel the Upkeep first. There is delay once an Upkeep has been cancelled before funds can be withdrawn. The number of blocks delay varies by network and once the delay has passed, you can Withdraw funds.

Interacting directly with the Chainlink Automation Registry

After registration, you can interact directly with the registry contract functions such as `cancelUpkeep` and `addFunds` using your Upkeep ID. The Registry Address might change when new contracts are deployed with new functionality.

```
# migrate-to-v2.mdx:
```

```
---
section: automation
date: Last Modified
title: "Migrate to v2.1"
---
```

```
import { Aside, ClickToZoom } from "@components"
```

Chainlink Automation 2.1 is a consensus-driven Automation solution that allows you to cut onchain gas costs by using cryptographically verified offchain compute. Automation 2.1 provides 10M gas worth of offchain compute, which is significantly more than previous versions. Additionally, Automation 2.1 provides increased reliability, performance, log trigger capability, and the ability to use `StreamsLookup` to retrieve Data Streams.

You can migrate most upkeeps that use Automation version 1.2 and later in the Chainlink Automation App or in the block scanner. When you migrate upkeeps through the registry, you retain the Upkeep ID. Before you migrate, read the migration checklist to maximize your benefits from Automation 2.1.

For upkeeps on older registry versions 1.0 (Ethereum Mainnet), and 1.1 (BNB Mainnet and Polygon Mainnet), you must migrate manually by cancelling and re-registering your upkeep in the Chainlink Automation App. After you do this manual migration, future migrations will be easier because your new upkeeps will be eligible for the simpler migration process.

`<Aside type="caution" title="Grant permission to the forwarder address">`
After you migrate, your new upkeep has a unique forwarder address to increase security. This address will be the unique `msg.sender` for your upkeep.

If your upkeep is restricted to a single address calling it, you must give permission to the forwarder address. Otherwise, Automation will no longer be able to execute your function.

`</Aside>`

Migrating using the Chainlink Automation App

The Chainlink Automation App offers a streamlined migration process for upkeeps using registry versions 1.2 and later. To migrate upkeeps with older versions, follow the manual migration process instead.

1. Navigate to the Chainlink Automation App, select the supported blockchain you're using, and connect your wallet.

```
<div class="remix-callout">
  <a href="https://automation.chain.link">Open the Chainlink Automation
App</a>
```

</div>

<ClickToZoom src="/images/automation/v2-migration/landing-page.png" />

1. To start migrating a specific upkeep, select the upkeep. In the Details page, expand the Actions menu and select Migrate upkeep.

<ClickToZoom src="/images/automation/v2-migration/upkeep-details.png" />

If you have multiple upkeeps to migrate, start the migration using the link in your upkeeps dashboard. This link displays only if you have one or more upkeeps to migrate:

<ClickToZoom src="/images/automation/v2-migration/1-bulk-migration-ui.png" />

1. Follow the prompts to approve and confirm the transactions in your wallet to migrate your upkeeps.

<ClickToZoom src="/images/automation/v2-migration/approve-migration.png" />

Upkeeps that are successfully migrated will show the following transaction logs:

<ClickToZoom src="/images/automation/v2-migration/migration-txn.png" />

1. If your upkeep restricts msg.sender to the previous registry address, update your contract to use the new forwarder address.

After the migration is complete:

- Your balance is transferred to the new registry automatically.
- The new forwarder address becomes available.
- Read the migration checklist to understand further updates you might need to make in your contracts.

Migrating upkeeps on paused registries

If you have any upkeeps that are not yet migrated to v2.1, using older registries that are paused, the only action you can take is either to migrate these upkeeps or to cancel them. Affected upkeeps will show a Deprecated label in the Chainlink Automation App. When you hover over this label, it displays a link you can click to begin the migration process for the upkeep:

<ClickToZoom src="/images/automation/v2-migration/ui-deprecation-label.png" />

Migrating upkeeps using block scanner

To migrate one or more upkeeps using the scanner:

1. Navigate to the blockscanner for the desired chain to the Automation registry containing your upkeeps. You can find the registry address in the Chainlink Automation App under Upkeep details.

1. Under Contract/Write contract expand the migrateUpkeeps function.

Enter a list of upkeep IDs. For example:

[99297446083125056953495436926707926113001699970195513612134585638496630959874,63026166139768298778579034679995711427803187091626268721992534052921779884688].

1. Enter the destination registry address, which is the latest registry address on this chain. You can find this address on the Supported Networks page, or at the top of the Chainlink Automation App with the desired chain selected.

1. Execute the migrateUpkeeps function and confirm the transaction. When this

transaction is confirmed, your upkeeps will be migrated to the latest registry, and each upkeep will have a unique forwarder address.

1. If your upkeep restricts msg.sender to the previous registry address, update your contract to use the new forwarder address.

After the migration is complete:

- Your balance is transferred to the new registry automatically.
- The new forwarder address becomes available.
- Read the migration checklist to understand further updates you might need to make in your contracts.

Migrating older upkeeps manually

For upkeeps on registry versions 1.0 and 1.1, you must migrate upkeeps manually:

1. Navigate to the Upkeep in the Chainlink Automation App.

```
<div class="remix-callout">
  <a href="https://automation.chain.link">Open the Chainlink Automation
App</a>
</div>
```

1. In the Details section, navigate to the center Upkeep card. Copy the Upkeep address - you need this for Step 5.

1. Expand the Actions menu and select Cancel upkeep.

1. Approve the transaction in your wallet. When this transaction is confirmed, you must wait 50 blocks before you can withdraw funds.

1. Return to the main Chainlink Automation App landing page. Register a new upkeep, providing the Upkeep address of your old upkeep.

1. If your upkeep restricts msg.sender to the previous registry address, update your contract to use the new forwarder address.

After migration, you have a new upkeep on Automation 2.1 with the same interface as your old upkeep. Future migrations are eligible for the simpler migration process.

After the migration is complete:

- Your balance is transferred to the new registry automatically.
- The new forwarder address becomes available.
- Read the migration checklist to understand further updates you might need to make in your contracts.

Update permissions

Your new upkeep has a new unique forwarder to increase security for your upkeep. This address will be the unique msg.sender for your upkeep. If your upkeep restricts msg.sender to the previous registry address, you must give permission to the forwarder address. Otherwise, Automation will no longer be able to execute your function.

1. The forwarder address becomes available after migrating your upkeep. You can find this in the Chainlink Automation App, within the upkeep's Details section:

```
<ClickToZoom src="/images/automation/v2-migration/forwarder-address.png" />
```

1. Update your contract to use the forwarder address by following the instructions on the Forwarder page.

Forwarders by upkeep type

This diagram shows the flow of different contracts that Automation 2.1 deploys

for new and migrated upkeeps. Compared to custom logic and log trigger upkeeps, time-based upkeeps have an additional contract:

```
<ClickToZoom src="/images/automation/v2-migration/v2-upkeeps-by-type.png" />
```

- For custom logic and log trigger upkeeps, the msg.sender in relation to your contract is the unique forwarder that Automation deploys when you migrate your upkeep.
- For time-based upkeeps, Automation deploys a unique forwarder and a unique CRON upkeep contract. In this case, the CRON upkeep contract is the msg.sender in relation to your contract.

Migration checklist

Before you migrate, be aware of several important changes listed here.

Unique forwarder

Automation 2.1 upkeeps are called from a unique forwarder per upkeep and not from the registry. If your upkeep restricts msg.sender to the previous registry address, you must update it to the newly created forwarder address. The forwarder address becomes available only after the upkeep has been migrated. This forwarder address will remain constant in future migrations.

Update programmatic upkeeps

Note that migration moves upkeeps from one registry to another. If you interact with your upkeep programatically using Solidity or other interfaces, you must update your code to make sure that you are referencing the correct registry and registrar for your migrated upkeeps:

- Update the registry and registrar addresses.
- Ensure you use the latest version of the ABI for the registry and registrar.

Get the latest ABI

The latest ABI for Automation 2.1 is in the @chainlink npm package:

- Registry ABI: @chainlink/contracts/abi/v0.8/IKeeperRegistryMaster.json
- Registrar ABI: @chainlink/contracts/abi/v0.8/AutomationRegistrar21.json

After updating to the latest ABI, you will be able to execute registry.getForwarder(upkeepID) to get the forwarder address in Solidity.

Check function signatures

If your contract makes function calls to the registry from your upkeep contract, follow the latest ABI.

Funding is moved with migration

When you migrate, your LINK funding is moved from one registry to the next automatically.

Migration questions and feedback

If you have questions or feedback, contact us in the #automation channel on the Chainlink Discord server.

```
# register-upkeep-in-contract.mdx:
```

```
---
```

```
section: automation
```

```

date: Last Modified
title: "Register Upkeeps Programmatically"
isMdx: true
whatsnext:
  {
    "Create Automation-Compatible Contracts":
"/chainlink-automation/guides/compatible-contracts",
    "Troubleshoot and Debug Upkeeps":
"/chainlink-automation/reference/debugging-errors",
  }
---

```

```
import { Aside, CodeSample } from "@components"
```

This guide explains how to register an upkeep from within your smart contract, also called programmatic upkeep creation. Your contract can then interact with it via the registry to get its balance, fund it, edit it, or cancel it using the upkeepID.

Before you begin

Before beginning this process, complete the following tasks:

1. Ensure the smart contract you want to automate is Automation Compatible. To learn more about the contracts Chainlink Automation uses, [click here](#).
1. Ensure you have sufficient LINK in the contract that will be registering the Upkeep. Use `faucets.chain.link` to get testnet LINK.
1. Ensure you have the addresses for the LINK token you are using, and the correct registry/registrar. You can find these values on the [Supported Networks](#) page. Note: You can retrieve the LINK token address by calling the function `getLinkAddress` on the registry .
1. Use variables for the registry and registrar addresses that your admin can change as new versions of Chainlink Automation are released.
1. The interface for LINK and Registrar for registration, interface for Registry for subsequent actions
1. Interface is like the API specification of interacting with the contract.

Register the upkeep

Programmatically registering an upkeep happens in two steps:

1. Call the LINK token to give allowance to the Automation registrar for the amount of LINK you will fund your upkeep with at registration time, e.g. `Pizza` code to do
1. Call `registerUpkeep` on the Registrar contract using the `RegistrationParams` struct. You will receive the `upkeepID` if successful.

Var type	Var Name	Example value	Description
String	name	"Test upkeep"	Name of upkeep that will be displayed in the UI.
bytes	encryptedEmail	0x	Can leave blank. If registering via UI we will encrypt email and store it here.
address	upkeepContract		Address of your Automation-compatible contract

uint32	gasLimit	500000	The maximum gas limit that will be used for your txns. Rather over-estimate gas since you only pay for what you use, while too low gas might mean your upkeep doesn't perform. Trade-off is higher gas means higher minimum funding requirement.
address	adminAddress		The address that will have admin rights for this upkeep. Use your wallet address, unless you want to make another wallet the admin.
uint8	triggerType	0 or 1	0 is Conditional upkeep, 1 is Log trigger upkeep
bytes	checkData	0x	checkData is a static input that you can specify now which will be sent into your checkUpkeep or checkLog, see interface.
bytes	triggerConfig	0x	The configuration for your upkeep. 0x for conditional upkeeps, or see next section for log triggers.
bytes	offchainConfig	0x	Leave as 0x, or use this field to set a gas price threshold for your upkeep. Must be a JSON object and CBOR encoded - see more details and examples on formatting.
uint96	amount	1000000000000000000	Ensure this is less than or equal to the allowance just given, and needs to be in WEI.

Upkeep registration parameters and examples

Depending on the trigger you are using, the triggerConfig will be different. Browse the triggers below to understand how to set up triggerConfig.

Custom logic upkeeps

Parameters

For upkeeps with triggers using onchain state only, the following parameters are needed:

Code sample

```
<CodeSample src="/samples/Automation/UpkeepIDConditionalExample.sol" />
```

Log trigger upkeeps

Parameters

For upkeeps with triggers using emitted logs, the following parameters are needed:

solidity

```
struct LogTriggerConfig {
    address contractAddress; // must have address that will be emitting the log
    uint8 filterSelector; // must have filterSelector, denoting which topics
    // apply to filter ex 000, 101, 111...only last 3 bits apply
    bytes32 topic0; // must have signature of the emitted event
    bytes32 topic1; // optional filter on indexed topic 1
    bytes32 topic2; // optional filter on indexed topic 2
    bytes32 topic3; // optional filter on indexed topic 3
}
```

where filterSelector is a bitmask mapping and value is set depending on the selection of filters

filterSelector	Topic 1	Topic 2	Topic 3	
----------------	---------	---------	---------	--

	-----	-----	-----	-----
0	Empty	Empty	Empty	
1	Filter	Empty	Empty	
2	Empty	Filter	Empty	
3	Filter	Filter	Empty	
4	Empty	Empty	Filter	
5	Filter	Empty	Filter	
6	Empty	Filter	Filter	
7	Filter	Filter	Filter	

Code sample

```
<CodeSample src="/samples/Automation/UpkeepIDlogTriggerExample.sol" />
```

```
# register-upkeep.mdx:
```

```
---
section: automation
date: Last Modified
title: "Register a Custom Logic Upkeep"
isMdx: true
whatsnext:
  {
    "Register Log Trigger Upkeeps": "/chainlink-automation/guides/log-trigger/",
    "Register Upkeeps Programmatically": "/chainlink-automation/guides/register-
upkeep-in-contract/",
    "Automation Interfaces": "/chainlink-automation/reference/automation-
interfaces",
    "Automation Architecture": "/chainlink-automation/concepts/automation-
architecture/",
  }
---
```

```
import { Aside, CodeSample, ClickToZoom } from "@components"
```

Create powerful automation for your smart contract that leverages custom logic to trigger specified actions. This guide explains how to register a custom logic upkeep that uses a compatible contract. You can register it using the Chainlink Automation App or from within a contract that you deploy.

```
{/ prettier-ignore /}
{" "}
```

```
<Aside type="tip" title="Testing and best practices">
  Follow the best practices when creating a compatible contract and test
  your upkeep on a testnet before deploying it to a mainnet.
</Aside>
```

Using the Chainlink Automation App

```
<div class="remix-callout">
  <a href="https://automation.chain.link">Open the Chainlink Automation App</a>
</div>
```

Click the Register New Upkeep button

```
{" "}
```

```
<ClickToZoom src="/images/automation/auto-ui-home.png" />
```

Connecting your wallet

If you do not already have a wallet connected with the Chainlink Automation

network, the interface will prompt you to do so. Click the Connect Wallet button and follow the remaining prompts to connect your wallet to one of the Automation supported blockchain networks.

```
<ClickToZoom src="/images/automation/auto-ui-wallet.png" />
```

Trigger selection

Select Custom Logic trigger.

```
{" "}
```

```
<ClickToZoom src="/images/automation/uiselecttrigger.png" />
```

Using custom logic triggers

Provide the address of your compatible contract. You do not need to verify the contract onchain, but it must be compatible with the AutomationCompatibleInterface contract.

Entering upkeep details

Provide the following information in the Automation app:

- Upkeep name: This will be publicly visible in the Chainlink Automation app.
- Gas limit: This is the maximum amount of gas that your transaction requires to execute on chain. This limit cannot exceed the performGasLimit value configured on the registry. Before the network executes your transaction on chain, it simulates the transaction. If the gas required to execute your transaction exceeds the gas limit that you specified, your transaction will not be confirmed. Developers also have the ability to update performGasLimit for an upkeep. Consider running your function on a testnet to see how much gas it uses before you select a gas limit. This can be changed afterwards.
- Starting balance (LINK): Specify a LINK starting balance to fund your upkeep. See the LINK Token Contracts page to find the correct contract address and access faucets for testnet LINK. This field is required. You must have LINK before you can use the Chainlink Automation service.

```
{/ prettier-ignore /}  
{" "}
```

```
<Aside type="tip" title="Funding Upkeep">  
  You should fund your contract with more LINK that you anticipate you will  
  need. The network will not check or  
  perform your Upkeep if your balance is too low based on current exchange  
  rates. View the Automation  
  economics page to learn more about the cost of using Chainlink  
  Automation.  
</Aside>
```

```
{/ prettier-ignore /}  
{" "}
```

```
<Aside type="tip" title="ERC-677 Link">  
  For funding on Mainnet, you need ERC-677 LINK. Many token bridges give you  
  ERC-20 LINK tokens. Use PegSwap to  
  convert Chainlink tokens (LINK) to be ERC-677 compatible. Use  
  faucets.chain.link to get testnet LINK.  
</Aside>
```

- Check data: This field is provided as an input for when your checkUpkeep function is simulated. Either leave this field blank or specify a hexadecimal value starting with 0x. To learn how to make flexible upkeeps using checkData, see the Flexible Upkeeps guide.

- Your email address (optional): This email address will be used to send you an email notification when your upkeep is underfunded.

Complete upkeep registration

Click Register upkeep and confirm the transaction in MetaMask.
!Upkeep Registration Success Message

Your upkeeps will be displayed in your list of Active Upkeeps. You must monitor the balance of your upkeep. If the balance drops below the minimum balance, the Chainlink Automation Network will not perform the Upkeep. See Managing Upkeeps to learn how to manage your upkeeps.

streams-lookup-error-handler.mdx:

```
---
section: automation
date: Last Modified
title: "Using the StreamsLookup error handler"
isMdx: true
whatsnext: { "Troubleshoot and Debug Upkeeps":
"/chainlink-automation/reference/debugging-errors",
"Automation Interfaces": "/chainlink-automation/reference/automation-
interfaces",
"Automation Contracts": "/chainlink-automation/reference/automation-contracts" }
---
```

```
import { Aside } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<Aside type="note" title="Data Streams Mainnet Early Access">
  Chainlink Data Streams is available on Arbitrum Mainnet and Arbitrum Sepolia.
</Aside>
```

```
<Aside type="note" title="Talk to an expert">
  Contact us to talk to an expert about integrating
  Chainlink Data Streams with your applications.
</Aside>
```

```
<DataStreams section="streamsLookupErrorHandler" />
```

streams-lookup.mdx:

```
---
section: automation
date: Last Modified
title: "Access Data Streams Using Automation"
isMdx: true
whatsnext: { "Troubleshoot and Debug Upkeeps":
"/chainlink-automation/reference/debugging-errors",
"Automation Interfaces": "/chainlink-automation/reference/automation-
interfaces",
"Automation Contracts": "/chainlink-automation/reference/automation-contracts" }
---
```

```
import { Aside, CodeSample } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<Aside type="note" title="Early Access">
  Data Streams is available on Arbitrum Mainnet and Arbitrum Sepolia in Early
  Access.{ " "}
  <a href="https://chainlinkcommunity.typeform.com/datastreams?"
```

[Contact us](#) to talk to an expert about integrating Chainlink Data Streams with your applications.

<DataStreams section="gettingStarted" />

Debugging StreamsLookup

Read our debugging section to learn how to identify and resolve common errors when using StreamsLookup.

automation-economics.mdx:

```
---
section: automation
date: Last Modified
title: "Automation Billing and Costs"
isMdx: true
whatsnext: { "Automation Architecture":
"/chainlink-automation/concepts/automation-architecture",
"Supported Networks for Automation": "/chainlink-automation/overview/supported-networks" }
---
```

Cost of using Chainlink Automation

Chainlink Automation only requires an execution fee for transactions onchain. This fee includes the transaction cost, a node operator percentage fee (refer to the formula below), and a small fixed gas overhead accounting for gas between the network and the registry. The percentage fee compensates the Automation Network for monitoring and performing your upkeep. The Automation percentage fee varies by chain and is listed on our Supported Networks page.

Formula for Registry v2.1

$$\text{Fee}_{\text{LINK}} = [\text{tx.gasPrice}_{\text{Native WEI}} \cdot (\text{gasUsed} + \text{gasOverhead}) \cdot (1 + \text{premium\%})] / [\text{LINK}_{\text{Native Rate in WEI}}]$$

On Automation v2.2 and earlier, there is a flat per-transaction fee of 0.01 LINK in the following cases:

- For mainnet transactions on Optimism and Base, to account for L1 transaction costs, ensuring that node operators are fairly compensated
- For testnet transactions on all supported networks, in order to account for node infrastructure costs

There is no registration fee or other fees for any offchain computation.

Fee calculation example

An upkeep transaction was performed on Polygon mainnet. It used 110,051 gas at a gas price of 182,723,799,380 wei. The node operator percentage on Polygon was 70% at the time of this transaction, and this fee varies by network. The LINK/POL exchange rate for this transaction was 7,308,290,731,273,610,000 wei. The upkeep's LINK balance was reduced by a fee of 0.008077 LINK. The preceding information and calculation can be found in the table below:

Variable	Description
Value	

tx.gasPrice _{Native WEI}	Gas price of the transaction

182, 723, 799,380		
gasUsed		Gas used for performUpkeep calculated in solidity
110,051		
gasOverhead		Fixed gas amount used for transaction call from node to Registry
80,000		
premium%		Current premium on Polygon which can be found on the Supported Networks page
LINK/Native_{Rate in WEI}		Exchange rate fetched from Chainlink Oracle
7,308,290,731,273,610,000		

0.008077 = [182,723,799,380 (110,051 + 80,000) (1 + 70%)]/[7,308,290,731,273,610,000]

How funding works

Upkeeps have a LINK (ERC-677) balance. Every time an onchain transaction is performed for your upkeep, its LINK balance will be reduced by the LINK fee.

Your upkeep's balance must exceed the minimum balance. If this requirement is not met, the Automation Network will not perform onchain transactions. You can add funds using the Chainlink Automation App or by directly calling the addFunds() function on the AutomationRegistry contract. Anyone can call the addFunds() function.

Withdrawing funds

To withdraw a LINK balance, you must cancel your upkeep first. Any upkeep that has not spent more than an aggregated amount of 0.1 LINK fees over the span of its lifetime is subject to a 0.1 LINK fee. This cancellation fee protects node operators from spammers who register jobs that never perform.

Example 1: Your upkeep has spent 4.8 LINK over its lifetime and has a balance of 5 LINK. When it is cancelled, I will receive 5 LINK.

Example 2: Your upkeep has spent 0 LINK over its lifetime and has a balance of 5 LINK. When it is cancelled, I will receive 4.9 LINK.

No node competition

Individual Automation Nodes do not compete with one another, but rather work together to ensure all registered upkeeps are performed. This makes costs more predictable upfront, enabling you to estimate costs based on the expected gas consumption.

Minimum balance

The Chainlink Automation Network is designed to perform your upkeep even when gas prices spike. The minimum balance in LINK reflects the best estimate of the cost to perform your upkeep when gas prices spike. To ensure your upkeep is monitored and performed, ensure that your upkeep's balance is above this minimum balance.

The minimum balance is calculated using the current fast gas price, the gas limit you entered for your upkeep, the max gas multiplier, and the LINK/Native_{Rate in WEI} for conversion to LINK. To find the latest value for the gasCeilingMultiplier, see the Registry Configuration page.

Follow maintain a minimum balance to ensure that your upkeep is funded.

Price selection and gas bumping

Automation Nodes select the gas price dynamically based on the prices of

transactions within the last several blocks. This optimizes the gas price based on current network conditions. Automation Nodes are configured to select a price based on a target percentile.

If the Automation Node does not see the performUpkeep transaction get confirmed within the next few blocks, it automatically replaces the transaction and bumps the gas price. This process repeats until the transaction is confirmed.

ERC-677 LINK

For funding on mainnet, you will need ERC-677 LINK. Many token bridges give you ERC-20 LINK tokens. Use PegSwap to convert Chainlink tokens (LINK) to be ERC-677 compatible. Use faucets.chain.link to get testnet LINK.

```
# automation-release-notes.mdx:
```

```
---
section: automation
date: Last Modified
title: "Chainlink Automation Release Notes"
isMdx: true
whatsnext:
  {
    "Register Time-Based Upkeeps": "/chainlink-automation/guides/job-scheduler",
    "Register Custom Logic Upkeeps": "/chainlink-automation/guides/register-
upkeep",
    "Register Log Trigger Upkeeps": "/chainlink-automation/guides/log-trigger/",
  }
---
```

```
import { ClickToZoom } from "@components"
```

Chainlink Automation Release Notes

- Log trigger upkeeps are generally available
- Migrating upkeeps on paused registries
- Automation on Base Sepolia
- Deprecation of older upkeeps
- Automation on Gnosis
- Registrar deprecation through v2.0
- Automation on Polygon Amoy
- Polygon testnet support
- Automation StreamsLookup error handler
- Automation debugging script
- Automation on Optimism Sepolia
- Automation on Base
- v2.0 release
- Automation on Optimism
- Chainlink Keepers is now Chainlink Automation
- v1.3 release
- v1.2 release
 - Manually migrating upkeeps from v1.1 to v1.2
- Underfunded upkeep email notifications
- Keepers on Fantom
- Keepers on Avalanche
- Keepers on Ethereum Rinkeby
- Keepers on Binance Smart Chain and Polygon
- Keepers v1.1 launch on Ethereum
- Questions

2024-08-29 - Log trigger upkeeps are generally available

Log trigger upkeeps are now generally available. Learn more about how Chainlink

Automation processes logs for log trigger upkeeps.

2024-07-30 - Migrating upkeeps on paused registries

If you have any upkeeps that are not yet migrated to v2.1, using older registries that are paused, the only action you can take is either to migrate these upkeeps or to cancel them. Affected upkeeps will show a Deprecated label in the Chainlink Automation App. When you hover over this label, it displays a link you can click to begin the migration process for the upkeep:

<ClickToZoom src="/images/automation/v2-migration/ui-deprecation-label.png" />

2024-06-28 - Automation on Base Sepolia

Chainlink Automation is live on Base Sepolia.

2024-06-24 - Deprecation of older upkeeps

Existing upkeeps on versions earlier than v2.1 will stop being performed on August 29, 2024.

Migrate your older upkeeps to the latest version of Automation.

Older Automation registrars through v2.0 have already been deprecated, so you can't register new upkeeps on versions earlier than v2.1.

2024-06-06 - Automation on Gnosis

Chainlink Automation is live on Gnosis.

2024-06-03 - Registrar deprecation through v2.0

Older Automation registrars for v1.0, v1.1, v1.2, v1.3, and v2.0 are deprecated on all supported networks (Ethereum, Avalanche, BSC, Polygon, Arbitrum and Optimism). You can no longer register new upkeeps using these older versions. Please migrate your older upkeeps to Automation 2.1 to ensure they remain operational as we start deprecating older versions.

2024-04-23 - Automation on Polygon Amoy

Chainlink Automation is live on Polygon Amoy.

2024-04-13 - Polygon testnet support

The Mumbai network has stopped producing blocks, so example code will not function on this network. Check again soon for updates about future testnet support on Polygon.

2024-03-07 - Automation StreamsLookup error handler

The Automation StreamsLookup error handler is available to help you handle potential errors with StreamsLookup upkeeps. When you add the new `checkErrorHandler` function, you can define custom logic to handle some errors offchain and handle other errors onchain in `performUpkeep`.

2024-02-27 - Automation on Optimism Sepolia

Chainlink Automation is live on Optimism Sepolia.

2024-02-27 - Automation debugging script

The Chainlink Automation debugging script is available to help you debug and diagnose possible issues with registered upkeeps in Automation 2.1 registries. The script can debug custom logic upkeeps, log trigger upkeeps, and upkeeps that use StreamsLookup.

2023-12-07 - Automation on Base

Chainlink Automation is live on Base.

2023-10-02 - v2.0 release

Automation 2.0 is now live on Ethereum, Binance Smart Chain, Polygon, Avalanche, Arbitrum, and Optimism. Automation 2.0 features include:

- Verifiable compute: The Automation DON now leverages a consensus mechanism, via Chainlink OCR3, to give you cryptographically verified compute. Save up to 90% of onchain gas costs by off-loading compute intensive tasks to the Automation DON.
- Log triggers: Natively use log data in your smart contracts with log triggers. Unlock new connection possibilities.
- StreamsLookup: Seamlessly access and use Chainlink's Low Latency Data in upkeeps via StreamsLookup. Build like the best Derivative protocols.
- Forwarder: A unique msg.Sender for your performUpkeep function so you can lock down sensitive upkeeps. Read more about the forwarder.

2023-05-15 - Automation on Optimism

Chainlink Automation is live on Optimism.

Chainlink Keepers is now Chainlink Automation

Chainlink Keepers has been renamed to Chainlink Automation. The table below describes what terms under the former Keepers naming system translate to under the current Automation naming system:

Keepers Term	Automation Term
Chainlink Keepers Network	Chainlink Automation Network
Keeper/Keeper Node	Automation/Automation Node
Keepers Job	Automation Job
Keepers Registry/Registrar	Automation Registry/Registrar
KeeperCompatible.sol	AutomationCompatible.sol
KeeperBase.sol	AutomationBase.sol
KeeperCompatibleInterface.sol	AutomationCompatibleInterface.sol

2022-09-23 - v1.3 Release

Keepers Registry v1.3 launched on Arbitrum Mainnet.

2022-08-04 - v1.2 Release

Keepers Registry v1.2 launched on Ethereum, Binance Smart Chain, Polygon, Avalanche, and Fantom

- Automatic upkeep registration approval: All upkeeps on mainnet are now automatically approved.
- Programmatic control: With automatic approval, you can now dynamically create, manage, and fund upkeeps from within your dApps and even have an upkeep fund itself. Learn more here.
- Advanced turn-taking algorithm: Our turn taking algorithm now supports upkeeps that require high-frequency execution.
- Durable ID and user-triggered migration: All upkeeps created in versions v1.2

and later will have durable IDs. v1.2 also supports user-triggered migration to future registry versions to make it easier to migrate to a new Keepers Registry and benefit from new features. Future migrations can still retain the existing ID. The ID is now a hash in format of a 77 digit integer.

- Configurable upkeeps: You can now edit the gas limit of your upkeep to easily customize your upkeep to fit your needs without having to create a new upkeep.

- Offchain compute improvements: The offchain compute sequence is improved for higher-fidelity representation of the gas and logic before transactions are submitted onchain. This helps to reduce reverts and reduce fees.

- Minimum spend requirement: As part of the mission to continuously enhance the security of the Chainlink Network for all participants, each registered upkeep will have a minimum spend requirement of 0.1 LINK, in aggregate across all transactions for the upkeep, to discourage network spam. Note that an upkeep is the automation job itself. It is not a transaction. Each upkeep can have thousands of transactions. If an upkeep has not spent more than 0.1 LINK across all transactions at the time of cancellation, then 0.1 LINK will be retained for the network. If more than 0.1 LINK has been spent by an upkeep, the full remaining balance of the upkeep will be withdrawable when the upkeep is canceled.

Manually migrating upkeeps from v1.1 to v1.2

If your upkeep ID has 77 digits, it is already migrated to v1.2 and no further action is required. If your upkeep ID has less than 4 digits, your upkeep is on the v1.1 registry. To migrate your upkeep from Keepers v1.1 to Keepers v1.2, you can cancel it in the Keepers App, and register an exact copy of the upkeep in the Keepers App. While you can see upkeeps from both v1.1 and v1.2 in the Keepers App, all new upkeeps in the Keepers App will be automatically created on Keepers v1.2.

2022-07-21 - Underfunded upkeep notifications

You will now receive notifications to the email address you register in your upkeep when your upkeep is underfunded. We are limiting notifications on the same upkeep to once per week.

2022-06-29 - Keepers on Fantom

Chainlink Keepers is live on the Fantom Network, Mainnet and Testnet.

2022-06-09 - Keepers on Avalanche

Chainlink Keepers is live on the Avalanche Network, Mainnet and Testnet.

2022-03-01 - Keepers on Ethereum Rinkeby

Chainlink Keepers is live on Ethereum Rinkeby.

2021-11-18 - Keepers on Binance Smart Chain and Polygon

Chainlink Keepers is live on the both Binance Smart Chain Mainnet and Testnet, and Polygon Mainnet and Mumbai testnet.

2021-08-05 - Keepers v1.1 launch on Ethereum

Chainlink Keepers officially launched on Ethereum Mainnet.

Questions

Ask questions in the #automation channel in our Discord server.

getting-started.mdx:

```
---
section: automation
date: Last Modified
title: "Getting Started with Chainlink Automation"
isMdx: true
whatsnext:
  {
    "Create Automation-Compatible Contracts":
"/chainlink-automation/guides/compatible-contracts",
    "Access Data Streams Using Automation":
"/chainlink-automation/guides/streams-lookup",
    "Automation Architecture": "/chainlink-automation/concepts/automation-
architecture",
    "Automation Billing and Costs": "/chainlink-automation/overview/automation-
economics",
  }
---
```

```
import { Address, Aside, ClickToZoom, CopyText } from "@components"
import { YouTube } from "@astro-community/astro-embed-youtube"
import { Tabs } from "@components/Tabs"
```

Chainlink Automation will reliably execute smart contract functions using a variety of triggers. Explore the examples below to see how Chainlink Automation works for each type of trigger. Before you begin, you will need an active cryptocurrency wallet such as Metamask.

- Time-based trigger: Use a time-based trigger to execute your function according to a time schedule.
- Custom logic trigger: Use a custom logic trigger to provide custom solidity logic that Automation Nodes evaluate (offchain) to determine when to execute your function onchain.
- Log trigger: Use log data as both trigger and input.

Try out Chainlink Automation

Click the tabs below to use Chainlink Automation with each type of trigger:

```
<Tabs client:visible>
  <Fragment slot="tab.timeBased">Time-based</Fragment>
  <Fragment slot="tab.customLogic">Custom Logic</Fragment>
  <Fragment slot="tab.logTrigger">Log Trigger</Fragment>
  <Fragment slot="panel.timeBased">
    Increment a counter every 5 minutes using our example contract.
```

1. Navigate to the Chainlink Automation app and connect to Arbitrum Sepolia in the top dropdown menu.

1. Connect your cryptocurrency wallet to the app if you haven't done so already. You may also need to fetch Arbitrum Sepolia testnet LINK [here](#).

1. Click Register new Upkeep and select Time-based trigger.

1. Under Target contract address, enter <Address contractUrl="https://sepolia.arbiscan.io/address/0x083935210524c0A8922ec610d1063Aa0A54d9d70" />. This is a simple counter contract that increments with each call. View the source code [here](#).

1. In the Contract call section, enter addInteger under Target function. In the Function inputs section, enter a number to increment by under intToAdd. Then click Next.

`<ClickToZoom
src="/images/automation/getting-started/gstbucontractaddress.png" />`

1. Specify the time schedule, for example every 5 minutes. Paste the cron expression `<CopyText text="/5" code/>` under Cron expression or select one of the example timers. Then click Next.

1. To learn more about CRON expressions, click here.

1. Enter an Upkeep name, your public key address under Admin Address, `<CopyText text="5000000" code/>` under Gas limit, and `<CopyText text="0.1" code/>` under Starting balance (LINK).

`<ClickToZoom
src="/images/automation/getting-started/gstbuupkeepdetails.png" />`

1. Click Register Upkeep.

1. After the transaction has completed, you can view the performs for your upkeep in the upkeep details.

You have successfully automated your first time-based upkeep. To learn more about creating time-based upkeeps, read here.

`</Fragment>`

`<Fragment slot="panel.customLogic">`

Increment a counter using custom logic stored onchain.

1. Navigate to the Chainlink Automation app and connect to Arbitrum Sepolia in the top dropdown menu.

1. Connect your cryptocurrency wallet to the app if you haven't done so already. You might also need to fetch LINK for the Arbitrum Sepolia testnet from faucets.chain.link.

1. Click Register new Upkeep and select Custom logic trigger.

1. Under Target contract address, enter `<Address
contractUrl="https://sepolia.arbiscan.io/address/0x6C0AAaEBCDb6F5D03759B8BF14b47BE491755530" />`. This contract is an Automation-compatible contract that uses logic stored onchain and onchain state to determine when to increment a counter. View the source code here. Click Next.

`<ClickToZoom
src="/images/automation/getting-started/gsclucontractaddress.png" />`

1. Enter an Upkeep name, your public key address under Admin Address, `<CopyText text="5000000" code/>` under Gas limit, and `<CopyText text="0.1" code/>` under Starting balance (LINK).

1. Finally, enter your public key address under Check data (Hexadecimal). `checkData` is optional static data that you can pass into your upkeep to ensure your counter increments.

1. Click Register Upkeep.

1. After the transaction is complete, you can view the performs for your upkeep in the upkeep details. Your upkeep should perform once every minute and stop after 4 performs.

`<ClickToZoom
src="/images/automation/getting-started/gscluupkeepdetails.png" />`

You have successfully automated your first custom logic upkeep. To learn more about creating custom logic upkeep, read [here](#).

```
</Fragment>
<Fragment slot="panel.logTrigger">
```

Increment an onchain counter using a log as trigger.

1. Navigate to the Chainlink Automation app and connect to Arbitrum Sepolia in the top dropdown menu.

1. Connect your cryptocurrency wallet to the app if you haven't done so already. You might also need to fetch LINK for the Arbitrum Sepolia testnet from faucets.chain.link.

1. Click Register new Upkeep and select Log trigger.

1. Under Contract to automate, enter `<Address contractUrl="https://sepolia.arbiscan.io/address/0xe817e4A71C69C72C01B31906F9F8591FbaB6b448" />`. This is a simple `iLogAutomation-compatible` example contract that increments a counter when a log is detected. View the [source code](#) here. Click Next.

1. Under Contract emitting logs, enter `<Address contractUrl="https://sepolia.arbiscan.io/address/0x1260206b960bB07F12d48C19fad505CeFc071bDd" />`. This is the contract Automation will listen to for emitted logs. View the [source code](#) here. Click Next.

```
<ClickToZoom
src="/images/automation/getting-started/gsltulogtriggerdetails.png" />
```

1. Under Emitted log select Bump from the dropdown menu. This is the log signature Automation will look for.

1. Log index topic filters are optional filters to narrow the logs you want to trigger your upkeep. For this example, enter your public key address under `addr` and leave the `num` field empty. Later when you call the `bump` function to emit the log, your `msg.Sender` address will be emitted in the log, triggering your upkeep. Click Next.

```
<ClickToZoom
src="/images/automation/getting-started/gsltuemitlogdetails.png" />
```

1. Enter an Upkeep name, your public key address under Admin Address, `<CopyText text="500000" code/>` under Gas limit, and `<CopyText text="0.1" code/>` under Starting balance (LINK).

1. Click Register Upkeep and wait for the transaction to complete.

1. To trigger your upkeep call `bump` on the trigger contract by navigating to the Arbitrum Sepolia scanner, connecting your wallet and executing the `bump` function. You can observe your upkeep's perform in the Automation dashboard.

You have successfully automated your first log trigger upkeep. To learn more about creating log trigger upkeep, read [here](#).

```
</Fragment>
</Tabs>
```

Supported networks and costs

For a list of blockchains that are supported by Chainlink Automation, see the [Supported Networks](#) page. To learn more about the cost of using Chainlink Automation, see the [Automation Economics](#) page.

Contact us

For help with your specific use case, contact us to connect with one of our Solutions Architects. You can also ask questions about Chainlink Automation on Stack Overflow or the #automation channel in our Discord server. Utility contracts can also help you get started quickly.

supported-networks.mdx:

```
---
section: automation
date: Last Modified
title: "Supported Blockchain Networks"
isMdx: true
whatsnext:
  {
    "Register Time-based Upkeeps": "/chainlink-automation/guides/job-scheduler",
    "Register Custom Logic Upkeeps": "/chainlink-automation/guides/register-
upkeep",
    "Register Log Trigger Upkeeps": "/chainlink-automation/guides/log-trigger",
    "Create Automation-Compatible Contracts":
"/chainlink-automation/guides/compatible-contracts",
    "Automation Billing and Costs": "/chainlink-automation/overview/automation-
economics",
  }
---
```

```
import { AutomationConfigList } from "@features/chainlink-automation"
import ResourcesCallout from
"@features/resources/callouts/ResourcesCallout.astro"
import CcipCommon from "@features/ccip/CcipCommon.astro"
import ChainlinkAutomation from
"@features/chainlink-automation/common/ChainlinkAutomation.astro"
```

```
<ChainlinkAutomation callout="deprecation" />
```

To use Chainlink Automation on certain networks, you may need to conduct token transfers. You can transfer tokens by using Chainlink CCIP or third-party applications such as XSwap.

```
<CcipCommon callout="thirdPartyApps" />
<ResourcesCallout callout="bridgeRisks" />
```

Parameters

- Payment Premium % (paymentPremiumPPB): This percentage premium compensates the Chainlink Automation Network for monitoring and performing your upkeep. Every time a transaction is submitted for your upkeep, your LINK balance is reduced by the transaction cost plus this percentage premium.
- Flat Fee Micro Link (flatFeeMicroLink): A flat fee charged per transaction on all testnets and Optimism Mainnet.
- Maximum Check Data Size (maxCheckDataSize): The maximum size, in bytes, that can be sent to your checkUpkeep function.
- Check Gas Limit (checkGasLimit): The maximum amount of gas that can be used by your checkUpkeep function for offchain computation.
- Perform Gas Limit (performGasLimit): The maximum amount of gas that can be used by the client contract's performUpkeep function for the onchain transaction. You can set an upper limit on your upkeep during registration, but this number must not exceed the maxPerformGas on the Registry.
- maximum Perform Data Size (maxPerformDataSize): The maximum size in bytes that can be sent to your performUpkeep function.
- Gas Ceiling Multiplier (gasCeilingMultiplier): Establishes a ceiling for the

maximum price based on the onchain fast gas feed.

- Minimum Upkeep Spend (LINK): The minimum amount of LINK an upkeep must spend over its lifetime. If the lifetime (or total) upkeep spend is below this amount, then at cancellation this amount will be held back.

Configurations

```
<AutomationConfigList />
```

```
# automation-contracts.mdx:
```

```
---
section: automation
date: Last Modified
title: "Automation Contracts"
isMdx: true
whatsnext: { "Automation Architecture":
"/chainlink-automation/concepts/automation-architecture",
"Automation Interfaces": "/chainlink-automation/reference/automation-interfaces"
}
---
```

Automation Nodes use the following contracts. You can find them in the Chainlink repository. For details about how to use them, visit the [Creating Compatible Contracts](#) guide. To understand the logic behind these contracts, visit the [Architecture](#) page.

- AutomationCompatible.sol: Imports the following contracts:
 - AutomationBase.sol: Enables the use of the cannotExecute modifier. Import this contract if you need for this modifier. See the checkUpkeep function for details.
 - AutomationCompatibleInterface.sol: The interface to be implemented in order to make your contract compatible. Import this contract for type safety.

AutomationRegistry.sol

AutomationRegistry21.sol: The registry contract that tracks all registered Upkeeps and the Automation Nodes that can perform them. Note: As Chainlink Automation continues adding new functionalities, a new Automation Registry is deployed and the contract address may change.

AutomationRegistrar.sol

AutomationRegistrar21.sol: The Registrar contract governs the registration of new Upkeeps on the associated AutomationRegistry contract. Users who want to register Upkeeps by directly calling the deployed contract have to call the Transfer-and-Call function on the respective ERC-677 LINK contract configured on the Registrar and ensure they pass the correct encoded function call and inputs.

UpkeepTranscoder.sol

UpkeepTranscode40.sol allows the conversion of upkeep data from previous Automation registry versions 1.2, 1.3, and 2.0 to registry 2.1.

AutomationForwarder.sol

AutomationForwarder.sol is a relay that sits between the registry and the customer's target contract. The purpose of the forwarder is to give customers a consistent address to authorize against that stays consistent between migrations. The Forwarder also exposes the registry address, so that users who want to programmatically interact with the registry can do so. The forward function in this contract is called by the registry and forwards the call to the target.

CronUpkeepFactory.sol

CronUpkeepFactory.sol serves as a delegate for all instances of CronUpkeep. Those contracts delegate their checkUpkeep calls onto this contract. Utilizing this pattern reduces the size of the CronUpkeep contracts. You can use this contract when creating a time-based upkeep programmatically. You can learn more about creating upkeeps programmatically [here](#).

```
# automation-interfaces.mdx:
```

```
---
section: automation
date: Last Modified
title: "Automation Interfaces"
isMdx: true
whatsnext:
  {
    "Automation Contracts": "/chainlink-automation/reference/automation-
contracts",
    "Troubleshoot and Debug Upkeeps":
"/chainlink-automation/reference/debugging-errors",
  }
---
```

```
import { CodeSample } from "@components"
import ChainlinkAutomation from
"@features/chainlink-automation/common/ChainlinkAutomation.astro"
```

Your Automation-compatible contracts may use the following interfaces. You can find them in the Chainlink repository. To understand how to implement these contracts, visit the [Compatible Contracts](#) page.

- If you want a log event to trigger your upkeep, use the ILogAutomation interface.
- If you want to use onchain state (excluding logs) in a custom calculation to trigger your upkeep, use AutomationCompatibleInterface interface.
- If you want to call a function just based on time, consider using a time-based upkeep.
- If you want to use Automation with Data Streams, use StreamsLookupCompatibleInterface interface.

ILogAutomation

```
<ChainlinkAutomation section="ilogautomation" />
```

AutomationCompatibleInterface

Custom logic upkeeps need to use the AutomationCompatibleInterface.sol interface. Click on one of the functions below to understand its parameters and limits.

Function Name	Description

checkUpkeep	Runs offchain to determine if the performUpkeep function should be called onchain.
performUpkeep	Contains the logic that should be executed onchain when checkUpkeep returns true.

checkUpkeep function

This view function contains the logic that runs offchain during every block as an ethcall to determine if performUpkeep should be executed onchain. To reduce onchain gas usage, attempt to do your gas intensive calculations offchain in checkUpkeep and pass the result to performUpkeep onchain. It is a best practice to import the AutomationCompatible.sol contract and use the cannotExecute modifier to ensure that the method can be used only for simulation purposes.

```
solidity
function checkUpkeep(
    bytes calldata checkData
) external view override returns (bool upkeepNeeded, bytes memory performData);
```

Below are the parameters and return values of the checkUpkeep function. Click each value to learn more about its design patterns and best practices:

Parameters:

- checkData: Fixed and specified at upkeep registration and used in every checkUpkeep. Can be empty (0x).

Return Values:

- upkeepNeeded: Boolean that when True will trigger the onchain performUpkeep call.
- performData: Bytes that will be used as input parameter when calling performUpkeep. If you would like to encode data to decode later, try abi.encode.

checkData

You can pass information into your checkUpkeep function from your upkeep registration to execute different code paths. For example, to check the balance on a specific address, set the checkData to abi encode the address. To learn how to create flexible upkeeps with checkData, please see our flexible upkeeps page.

<CodeSample src="snippets/Automation/checkData.sol" />

Tips on using checkData:

- Managing unbounded upkeeps: Limit the problem set of your onchain execution by creating a range bound for your upkeep to check and perform. This allows you to keep within predefined gas limits, which creates a predictable upper bound gas cost on your transactions. Break apart your problem into multiple upkeep registrations to limit the scope of work.

Example: You could create an upkeep for each subset of addresses that you want to service. The ranges could be 0 to 49, 50 to 99, and 100 to 149.

- Managing code paths: Pass in data to your checkUpkeep to make your contract logic go down different code paths. This can be used in creative ways based on your use case needs.

Example: You could support multiple types of upkeep within a single contract and pass a function selector through the checkData function.

performData

The response from checkUpkeep is passed to the performUpkeep function as performData. This allows you to perform complex and gas intensive calculations as a simulation offchain and only pass the needed data onchain.

You can create a highly flexible offchain computation infrastructure that can perform precise actions onchain by using checkData and performData. Both of

these computations are entirely programmable.

performUpkeep function for custom logic triggers

When checkUpkeep returns upkeepNeeded == true, the Automation node broadcasts a transaction to the blockchain to execute your performUpkeep function onchain with performData as an input.

Ensure that your performUpkeep is idempotent. Your performUpkeep function should change state such that checkUpkeep will not return true for the same subset of work once said work is complete. Otherwise the Upkeep will remain eligible and result in multiple performances by the Chainlink Automation Network on the exactly same subset of work. As a best practice, always check conditions for your upkeep at the start of your performUpkeep function.

solidity

```
function performUpkeep(bytes calldata performData) external override;
```

Parameters:

- performData: Data which was passed back from the checkData simulation. If it is encoded, it can easily be decoded into other types by calling abi.decode. This data should always be validated against the contract's current state.

performData

You can perform complex and broad offchain computation, then execute onchain state changes on a subset that meets your conditions. This can be done by passing the appropriate inputs within performData based on the results from your checkUpkeep. This pattern can greatly reduce your onchain gas usage by narrowing the scope of work intelligently in your own Solidity code.

- Identify a list of addresses that require work: You might have a number of addresses that you are validating for conditions before your contract takes an action. Doing this onchain can be expensive. Filter the list of addresses by validating the necessary conditions within your checkUpkeep function. Then, pass the addresses that meet the condition through the performData function.

For example, if you have a "top up" contract that ensures several hundred account balances never decrease below a threshold, pass the list of accounts that meet the conditions so that the performUpkeep function validates and tops up only a small subset of the accounts.

- Identify the subset of states that must be updated: If your contract maintains complicated objects such as arrays and structs, or stores a lot of data, you should read through your storage objects within your checkUpkeep and run your proprietary logic to determine if they require updates or maintenance. After that is complete, you can pass the known list of objects that require updates through the performData function.

StreamsLookupCompatibleInterface

```
<ChainlinkAutomation section="streamslookup" />
```

debugging-errors.mdx:

section: automation

date: Last Modified

title: "Debugging and Troubleshooting Upkeeps"

isMdx: true

whatsnext:

{

"Build Flexible Smart Contracts Using Automation":

```
"/chainlink-automation/guides/flexible-upkeeps",
  "Manage your Upkeeps": "/chainlink-automation/guides/manage-upkeeps",
}
---
```

import { ClickToZoom } from "@components"

Given an upkeep ID, this page contains different methods of understanding and fixing issues with Upkeeps.

Automation debugging script

You can use the Automation debugging script to debug and diagnose possible issues with registered upkeeps in Automation 2.1 registries. The script can debug custom logic upkeeps, log trigger upkeeps, and upkeeps that use StreamsLookup.

Underfunded upkeeps

In the Chainlink Automation app, you can see the registration details of the upkeep, alongside the balance required and performUpkeep history. If the upkeep is underfunded, you will see a warning on top of the page. Underfunded upkeeps will not be performed.

```
<ClickToZoom src="/images/automation/debugging-ui.png" />
```

Insufficient perform gas

If your performGasLimit is too low, the Automation Network will not execute your upkeep as it will fail in simulation. Consider increasing your perform upkeep gas limit in the UI. See supported Blockchain Networks for limits.

Insufficient check gas

If the amount of computation in your checkUpkeep exceeds our checkGasLimit, your upkeep will not be performed. You will have to reduce the amount of compute in checkUpkeep to bring the gas below the applicable limits. See supported Blockchain Networks for limits.

Paused upkeeps

If your upkeep is paused, your upkeep will not be performed. Please unpause it in the Chainlink Automation app.

StreamsLookup

Upkeep has not been allowlisted

Once you registered your upkeep, you need to ask the Data Streams team to allowlist your upkeepID and specify the feeds you will need to access. If your upkeepID has not been added to the allow list it will not perform an upkeep.

Requesting multiple feeds where one is not valid

It is possible to request multiple feeds by specifying the feeds in a string array. However, if one of the reports is invalid or not available then Automation will not return any values to your checkCallback function. This is to ensure correct execution of your contract.

StreamsLookup ErrorHandler

Handle the StreamsLookup upkeep error codes by using the StreamsLookup ErrorHandler.

Etherscan

You can view the registry or user's upkeep address on Etherscan to view transaction history. There are three types of information you can find on Etherscan:

- All transactions on registry: This shows all the performUpkeeps on the registry. You can view a specific performUpkeep transaction to debug more.
- Specific performUpkeep transaction: By diving deep in the logs, you can check the upkeep ID within the UpkeepPerformed log.
- Target's internal transactions: For a particular target contract, you can view its internal transactions which contains performUpkeep transactions for that contract by using the registry address as the filter for from address. Note: internal transactions are only visible on Etherscan.

Tenderly

You can use Tenderly to simulate checkUpkeep and/or performUpkeep on different blocks. Before following the steps below, make sure you have a Tenderly account.

1. Enter the address of your selected registry. You can find this on the Supported Networks page.
1. Select your network.
1. Click Fetch ABI to automatically fetch the registry ABI.
1. Select the checkUpkeep function or performUpkeep function.
1. Enter the ID of your Upkeep. You can find this in the Chainlink Automation app.
1. You can either enter a block number to simulate a past action or use a pending block number to view the current state of an action and view the end result of an of an action.
1. Once the simulation is complete, you will see the result. This will be either a success or an error. To understand errors, view information under the Debug tab. Note: if the performUpkeep is failing while the check is succeeding, Chainlink Automation will not broadcast transactions.

```
# counting-dnft.mdx:
```

```
---
section: automation
date: Last Modified
title: "Counting dNFT"
whatsnext:
  {
    "Automate the Batch Reveal of Collections": "/quickstarts/batch-reveal",
    "Create Dynamic NFTs": "/quickstarts/dynamic-metadata",
    "Automate Top-Up for Contract Balances": "/quickstarts/eth-balance-monitor",
    "Automation Top-Up for VRF Subscriptions": "/quickstarts/vrf-subscription-
monitor",
  }
---
```

```
import { Aside, CodeSample } from "@components"
```

View the template here.

This repository houses an example that automates counting with a dynamic SVG. The main contract is counting-svg.sol which is automated to create an entirely onchain SVG.

```
solidity
// SPDX-License-Identifier: MIT
// An example of a consumer contract that relies on a subscription for funding.
pragma solidity 0.8.17;
```

```

// Imports
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Counters.sol";
import "@openzeppelin/contracts/utils/Strings.sol";
import "@openzeppelin/contracts/utils/Base64.sol";

contract CountSVG is ERC721, ERC721URIStorage, Ownable {
    // Setup counters, we use the OpenZeppelin Counters library
    using Counters for Counters.Counter;
    Counters.Counter private tokenIdCounter;

    // Initialize the counter to 0
    uint256 count = 0;

    // Constructor for the contract
    constructor() ERC721("Counting SVG", "cSVG") {}

    // Mint a new NFT and update the URI
    function safeMint(address to) public onlyOwner {
        uint256 tokenId = tokenIdCounter.current();
        tokenIdCounter.increment();
        safeMint(to, tokenId);
        updateURI();
    }

    // Update the URI for the NFT
    function updateURI() internal {
        // Build the SVG
        string memory finalSVG = buildSVG();
        // Encode the SVG and add the metadata
        string memory json = Base64.encode(
            bytes(
                string(
                    abi.encodePacked(
                        '{"name": "Counting SVG",',
                        '"description": "An Automated Counting SVG",',
                        '"image": "data:image/svg+xml;base64,',
                        Base64.encode(bytes(finalSVG)),
                        '"}'
                    )
                )
            )
        );
        // Set the URI string
        string memory finalTokenURI =
string(abi.encodePacked("data:application/json;base64,", json));
        // Update the URI
        // NOTE: This is hardcoded to the first SVG only
        setTokenURI(0, finalTokenURI);
    }

    // Create the SVG
    function buildSVG() internal view returns (string memory) {
        string
            memory headSVG = "<svg xmlns='http://www.w3.org/2000/svg' version='1.1'
xmlns:xlink='http://www.w3.org/1999/xlink' xmlns:svgjs='http://svgjs.com/svgjs'
width='500' height='500' preserveAspectRatio='none' viewBox='0 0 500 500'> <g
clip-path='url(&quot;#SvgjsClipPath1094&quot;)' fill='none'> <rect width='500'
height='500' x='0' y='0' fill='#32325d'></rect> <circle r='23.56' cx='117.85'
cy='66.24' fill='url(#SvgjsLinearGradient1095)'></circle> <circle r='23.145'
cx='233.4' cy='16.22' fill='url(#SvgjsLinearGradient1096)'></circle> <circle

```

```

r='16.155' cx='57.91' cy='279.29' fill='url(#SvgjsLinearGradient1097)'></circle>
<circle r='29.12' cx='175.64' cy='2.15'
fill='url(#SvgjsLinearGradient1098)'></circle> <circle r='33.07' cx='423.83'
cy='387.89' fill='url(#SvgjsLinearGradient1099)'></circle> <circle r='33.35'
cx='296.87' cy='307.98' fill='url(#SvgjsLinearGradient1100)'></circle> <circle
r='31.39' cx='273.7' cy='61.31' fill='url(#SvgjsLinearGradient1101)'></circle>
<circle r='48.695' cx='108.9' cy='421.22'
fill='url(#SvgjsLinearGradient1102)'></circle> </g> <defs> <clipPath
id='SvgjsClipPath1094'> <rect width='500' height='500' x='0' y='0'></rect>
</clipPath> <linearGradient x1='70.72999999999999' y1='66.24'
x2='164.96999999999997' y2='66.24' gradientUnits='userSpaceOnUse'
id='SvgjsLinearGradient1095'> <stop stop-color='#e298de' offset='0.1'></stop>
<stop stop-color='#484687' offset='0.9'></stop> </linearGradient>
<linearGradient x1='187.11' y1='16.22' x2='279.69' y2='16.22'
gradientUnits='userSpaceOnUse' id='SvgjsLinearGradient1096'> <stop stop-
color='#32325d' offset='0.1'></stop> <stop stop-color='#424488'
offset='0.9'></stop> </linearGradient> <linearGradient x1='25.599999999999994'
y1='279.29' x2='90.22' y2='279.29' gradientUnits='userSpaceOnUse'
id='SvgjsLinearGradient1097'> <stop stop-color='#32325d' offset='0.1'></stop>
<stop stop-color='#424488' offset='0.9'></stop> </linearGradient>
<linearGradient x1='117.39999999999998' y1='2.1499999999999986' x2='233.88'
y2='2.1499999999999986' gradientUnits='userSpaceOnUse'
id='SvgjsLinearGradient1098'> <stop stop-color='#e298de' offset='0.1'></stop>
<stop stop-color='#484687' offset='0.9'></stop> </linearGradient>
<linearGradient x1='357.69' y1='387.89' x2='489.97' y2='387.89'
gradientUnits='userSpaceOnUse' id='SvgjsLinearGradient1099'> <stop stop-
color='#32325d' offset='0.1'></stop> <stop stop-color='#424488'
offset='0.9'></stop> </linearGradient> <linearGradient x1='230.17000000000002'
y1='307.98' x2='363.57000000000005' y2='307.98' gradientUnits='userSpaceOnUse'
id='SvgjsLinearGradient1100'> <stop stop-color='#84b6e0' offset='0.1'></stop>
<stop stop-color='#464a8f' offset='0.9'></stop> </linearGradient>
<linearGradient x1='210.92' y1='61.31' x2='336.48' y2='61.31'
gradientUnits='userSpaceOnUse' id='SvgjsLinearGradient1101'> <stop stop-
color='#32325d' offset='0.1'></stop> <stop stop-color='#424488'
offset='0.9'></stop> </linearGradient> <linearGradient x1='11.510000000000005'
y1='421.22' x2='206.29000000000002' y2='421.22' gradientUnits='userSpaceOnUse'
id='SvgjsLinearGradient1102'> <stop stop-color='#84b6e0' offset='0.1'></stop>
<stop stop-color='#464a8f' offset='0.9'></stop> </linearGradient> </defs>";

```

```

    string memory tailSVG = "</svg>";
    string memory bodySVG = string(
        abi.encodePacked(
            "<text x='50%' y='50%' fill='white' font-size='128' dominant-
baseline='middle' text-anchor='middle'>",
            Strings.toString(count),
            "</text>"
        )
    );

    // Concatenate the SVG parts
    string memory finalSVG = string(abi.encodePacked(headSVG, bodySVG,
tailSVG));
    return finalSVG;
}

// Increment the counter
function addToCount() public {
    count = count + 1;
    updateURI();
}

function burn(uint256 tokenId) internal override(ERC721, ERC721URIStorage) {
    super.burn(tokenId);
}

```

```

    function tokenURI(uint256 tokenId) public view override(ERC721,
ERC721URIStorage) returns (string memory) {
        return super.tokenURI(tokenId);
    }
}

```

vault-harvester.mdx:

```

---
section: automation
date: Last Modified
title: "Vault Harvester"
whatsnext:
    {
        "Automate the Batch Reveal of Collections": "/quickstarts/batch-reveal",
        "Create Dynamic NFTs": "/quickstarts/dynamic-metadata",
        "Automate Top-Up for Contract Balances": "/quickstarts/eth-balance-monitor",
        "Automation Top-Up for VRF Subscriptions": "/quickstarts/vrf-subscription-
monitor",
    }
---

```

```
import { Aside, CodeSample } from "@components"
```

Harvesting and compounding is key to maximize yield in DeFi yield aggregators. Automate harvesting and compounding using Chainlink Automation's decentralized automation network.

View the template [here](#).

Below is the main contract KeeperCompatibleHarvester.sol:

```

solidity
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "../libraries/UpkeepLibrary.sol";
import "../interfaces/IKeeperRegistry.sol";
import "../interfaces/IHarvester.sol";

abstract contract KeeperCompatibleHarvester is IHarvester, Ownable {
    using SafeERC20 for IERC20;

    // Contracts.
    IKeeperRegistry public keeperRegistry;

    // Configuration state variables.
    uint256 public performUpkeepGasLimit;
    uint256 public performUpkeepGasLimitBuffer;
    uint256 public vaultHarvestFunctionGasOverhead; // Estimated average gas cost
of calling harvest()
    uint256 public keeperRegistryGasOverhead; // Gas cost of upstream contract
that calls performUpkeep(). This is a private variable on KeeperRegistry.
    uint256 public chainlinkUpkeepTxPremiumFactor; // Tx premium factor/multiplier
scaled by 1 gwei (109).
    address public callFeeRecipient;

    // State variables that will change across upkeeps.

```

```

uint256 public startIndex;

constructor(
    address keeperRegistry,
    uint256 performUpkeepGasLimit,
    uint256 performUpkeepGasLimitBuffer,
    uint256 vaultHarvestFunctionGasOverhead,
    uint256 keeperRegistryGasOverhead
) {
    // Set contract references.
    keeperRegistry = IKeeperRegistry(keeperRegistry);

    // Initialize state variables from initialize() arguments.
    performUpkeepGasLimit = performUpkeepGasLimit;
    performUpkeepGasLimitBuffer = performUpkeepGasLimitBuffer;
    vaultHarvestFunctionGasOverhead = vaultHarvestFunctionGasOverhead;
    keeperRegistryGasOverhead = keeperRegistryGasOverhead;

    // Initialize state variables derived from initialize() arguments.
    (, OnchainConfig memory config, , , ) = keeperRegistry.getState();
    chainlinkUpkeepTxPremiumFactor = uint256(config.paymentPremiumPPB);
}

/
/ checkUpkeep /
/

function checkUpkeep(
    bytes calldata checkData // unused
)
    external
    view
    override
    returns (
        bool upkeepNeeded,
        bytes memory performData // array of vaults +
    )
{
    checkData; // dummy reference to get rid of unused parameter warning

    // get vaults to iterate over
    address[] memory vaults = getVaultAddresses();

    // count vaults to harvest that will fit within gas limit
    (HarvestInfo[] memory harvestInfo, uint256 numberOfVaultsToHarvest, uint256
newStartIndex) = countVaultsToHarvest(
        vaults
    );
    if (numberOfVaultsToHarvest == 0) return (false, bytes("No vaults to
harvest"));

    (address[] memory vaultsToHarvest, uint256 heuristicEstimatedTxCost, uint256
callRewards) = buildVaultsToHarvest(
        vaults,
        harvestInfo,
        numberOfVaultsToHarvest
    );

    uint256 nonHeuristicEstimatedTxCost =
calculateExpectedTotalUpkeepTxCost(numberOfVaultsToHarvest);

    performData = abi.encode(
        vaultsToHarvest,
        newStartIndex,

```

```

        heuristicEstimatedTxCost,
        nonHeuristicEstimatedTxCost,
        callRewards
    );

    return (true, performData);
}

function buildVaultsToHarvest(
    address[] memory vaults,
    HarvestInfo[] memory willHarvestVaults,
    uint256 numberOfVaultsToHarvest
)
    internal
    view
    returns (address[] memory vaultsToHarvest, uint256 heuristicEstimatedTxCost,
uint256 totalCallRewards)
{
    uint256 vaultPositionInArray;
    vaultsToHarvest = new address[](numberOfVaultsToHarvest);

    // create array of vaults to harvest. Could reduce code duplication from
countVaultsToHarvest via a another function parameter called loopPostProcess
    for (uint256 offset; offset < vaults.length; ++offset) {
        uint256 vaultIndexToCheck = UpkeepLibrary.getCircularIndex(startIndex,
offset, vaults.length);
        address vaultAddress = vaults[vaultIndexToCheck];

        HarvestInfo memory harvestInfo = willHarvestVaults[offset];

        if (harvestInfo.willHarvest) {
            vaultsToHarvest[vaultPositionInArray] = vaultAddress;
            heuristicEstimatedTxCost += harvestInfo.estimatedTxCost;
            totalCallRewards += harvestInfo.callRewardsAmount;
            vaultPositionInArray += 1;
        }

        // no need to keep going if we're past last index
        if (vaultPositionInArray == numberOfVaultsToHarvest) break;
    }

    return (vaultsToHarvest, heuristicEstimatedTxCost, totalCallRewards);
}

function countVaultsToHarvest(
    address[] memory vaults
) internal view returns (HarvestInfo[] memory harvestInfo, uint256
numberOfVaultsToHarvest, uint256 newStartIndex) {
    uint256 gasLeft = calculateAdjustedGasCap();
    uint256 vaultIndexToCheck; // hoisted up to be able to set newStartIndex
    harvestInfo = new HarvestInfo[](vaults.length);

    // count the number of vaults to harvest.
    for (uint256 offset; offset < vaults.length; ++offset) {
        // startIndex is where to start in the vaultRegistry array, offset is
position from start index (in other words, number of vaults we've checked so
far),
        // then modulo to wrap around to the start of the array, until we've
checked all vaults, or break early due to hitting gas limit
        // this logic is contained in getCircularIndex()
        vaultIndexToCheck = UpkeepLibrary.getCircularIndex(startIndex, offset,
vaults.length);
        address vaultAddress = vaults[vaultIndexToCheck];

```

```

        (bool willHarvest, uint256 estimatedTxCost, uint256 callRewardsAmount) =
willHarvestVault(vaultAddress);

        if (willHarvest && gasLeft >= vaultHarvestFunctionGasOverhead) {
            gasLeft -= vaultHarvestFunctionGasOverhead;
            numberOfVaultsToHarvest += 1;
            harvestInfo[offset] = HarvestInfo(true, estimatedTxCost,
callRewardsAmount);
        }

        if (gasLeft < vaultHarvestFunctionGasOverhead) {
            break;
        }
    }

    newStartIndex = UpkeepLibrary.getCircularIndex(vaultIndexToCheck, 1,
vaults.length);

    return (harvestInfo, numberOfVaultsToHarvest, newStartIndex);
}

function willHarvestVault(address vaultAddress) internal view returns (bool
willHarvestVault, uint256, uint256) {
    (bool shouldHarvestVault, uint256 estimatedTxCost, uint256 callRewardAmount)
= shouldHarvestVault(vaultAddress);
    bool canHarvestVault = canHarvestVault(vaultAddress);

    willHarvestVault = canHarvestVault && shouldHarvestVault;

    return (willHarvestVault, estimatedTxCost, callRewardAmount);
}

function canHarvestVault(address vaultAddress) internal view virtual returns
(bool canHarvest);

function shouldHarvestVault(
    address vaultAddress
) internal view virtual returns (bool shouldHarvestVault, uint256
txCostWithPremium, uint256 callRewardAmount);

/
/ performUpkeep /
/

function performUpkeep(bytes calldata performData) external override {
    (
        address[] memory vaultsToHarvest,
        uint256 newStartIndex,
        uint256 heuristicEstimatedTxCost,
        uint256 nonHeuristicEstimatedTxCost,
        uint256 estimatedCallRewards
    ) = abi.decode(performData, (address[], uint256, uint256, uint256,
uint256));

    runUpkeep(
        vaultsToHarvest,
        newStartIndex,
        heuristicEstimatedTxCost,
        nonHeuristicEstimatedTxCost,
        estimatedCallRewards
    );
}

function runUpkeep(

```

```

    address[] memory vaults,
    uint256 newStartIndex,
    uint256 heuristicEstimatedTxCost,
    uint256 nonHeuristicEstimatedTxCost,
    uint256 estimatedCallRewards
) internal {
    // Make sure estimate looks good.
    if (estimatedCallRewards < nonHeuristicEstimatedTxCost) {
        emit HeuristicFailed(
            block.number,
            heuristicEstimatedTxCost,
            nonHeuristicEstimatedTxCost,
            estimatedCallRewards
        );
    }

    uint256 gasBefore = gasleft();
    // multi harvest
    require(vaults.length > 0, "No vaults to harvest");
    (uint256 numberOfSuccessfulHarvests, uint256 numberOfFailedHarvests, uint256
calculatedCallRewards) = multiHarvest(
        vaults
    );

    // ensure newStartIndex is valid and set startIndex
    uint256 vaultCount = getVaultAddresses().length;
    require(newStartIndex >= 0 && newStartIndex < vaultCount, "newStartIndex out
of range.");
    startIndex = newStartIndex;

    uint256 gasAfter = gasleft();
    uint256 gasUsedByPerformUpkeep = gasBefore - gasAfter;

    // split these into their own functions to avoid Stack too deep
    reportProfitSummary(
        gasUsedByPerformUpkeep,
        nonHeuristicEstimatedTxCost,
        estimatedCallRewards,
        calculatedCallRewards
    );
    reportHarvestSummary(newStartIndex, gasUsedByPerformUpkeep,
numberOfSuccessfulHarvests, numberOfFailedHarvests);
}

function reportHarvestSummary(
    uint256 newStartIndex,
    uint256 gasUsedByPerformUpkeep,
    uint256 numberOfSuccessfulHarvests,
    uint256 numberOfFailedHarvests
) internal {
    emit HarvestSummary(
        block.number,
        // state variables
        startIndex,
        newStartIndex,
        // gas metrics
        tx.gasprice,
        gasUsedByPerformUpkeep,
        // summary metrics
        numberOfSuccessfulHarvests,
        numberOfFailedHarvests
    );
}

```



```

function reportProfitSummary(
    uint256 gasUsedByPerformUpkeep,
    uint256 nonHeuristicEstimatedTxCost,
    uint256 estimatedCallRewards,
    uint256 calculatedCallRewards
) internal {
    uint256 estimatedTxCost = nonHeuristicEstimatedTxCost; // use nonHeuristic
here as its more accurate
    uint256 estimatedProfit =
UpkeepLibrary.calculateProfit(estimatedCallRewards, estimatedTxCost);

    uint256 calculatedTxCost =
calculateTxCostWithOverheadWithPremium(gasUsedByPerformUpkeep);
    uint256 calculatedProfit =
UpkeepLibrary.calculateProfit(calculatedCallRewards, calculatedTxCost);

    emit ProfitSummary(
        // predicted values
        estimatedTxCost,
        estimatedCallRewards,
        estimatedProfit,
        // calculated values
        calculatedTxCost,
        calculatedCallRewards,
        calculatedProfit
    );
}

function multiHarvest(
    address[] memory vaults
)
    internal
    returns (uint256 numberOfSuccessfulHarvests, uint256 numberOfFailedHarvests,
uint256 cumulativeCallRewards)
{
    bool[] memory isSuccessfulHarvest = new bool[](vaults.length);
    for (uint256 i = 0; i < vaults.length; ++i) {
        (bool didHarvest, uint256 callRewards) = harvestVault(vaults[i]);
        // Add rewards to cumulative tracker.
        if (didHarvest) {
            isSuccessfulHarvest[i] = true;
            cumulativeCallRewards += callRewards;
        }
    }
}

(address[] memory successfulHarvests, address[] memory failedHarvests) =
getSuccessfulAndFailedVaults(
    vaults,
    isSuccessfulHarvest
);

emit SuccessfulHarvests(block.number, successfulHarvests);
emit FailedHarvests(block.number, failedHarvests);

numberOfSuccessfulHarvests = successfulHarvests.length;
numberOfFailedHarvests = failedHarvests.length;
return (numberOfSuccessfulHarvests, numberOfFailedHarvests,
cumulativeCallRewards);
}

function harvestVault(address vault) internal virtual returns (bool
didHarvest, uint256 callRewards);

function getSuccessfulAndFailedVaults(

```

```

    address[] memory vaults,
    bool[] memory.isSuccessfulHarvest
) internal pure returns (address[] memory successfulHarvests, address[] memory
failedHarvests) {
    uint256 successfulCount;
    for (uint256 i = 0; i < vaults.length; i++) {
        if (isSuccessfulHarvest[i]) {
            successfulCount += 1;
        }
    }

    successfulHarvests = new address[](successfulCount);
    failedHarvests = new address[](vaults.length - successfulCount);
    uint256 successfulHarvestsIndex;
    uint256 failedHarvestIndex;
    for (uint256 i = 0; i < vaults.length; i++) {
        if (isSuccessfulHarvest[i]) {
            successfulHarvests[successfulHarvestsIndex++] = vaults[i];
        } else {
            failedHarvests[failedHarvestIndex++] = vaults[i];
        }
    }

    return (successfulHarvests, failedHarvests);
}

/      /
/ Set  /
/      /

function setPerformUpkeepGasLimit(uint256 performUpkeepGasLimit) external
override onlyOwner {
    performUpkeepGasLimit = performUpkeepGasLimit;
}

function setPerformUpkeepGasLimitBuffer(uint256 performUpkeepGasLimitBuffer)
external override onlyOwner {
    performUpkeepGasLimitBuffer = performUpkeepGasLimitBuffer;
}

function setHarvestGasConsumption(uint256 harvestGasConsumption) external
override onlyOwner {
    vaultHarvestFunctionGasOverhead = harvestGasConsumption;
}

/      /
/ View /
/      /

function getVaultAddresses() internal view virtual returns (address[] memory);

function getVaultHarvestGasOverhead(address vault) internal view virtual
returns (uint256);

function calculateAdjustedGasCap() internal view returns (uint256
adjustedPerformUpkeepGasLimit) {
    return performUpkeepGasLimit - performUpkeepGasLimitBuffer;
}

function calculateTxCostWithPremium(uint256 gasOverhead) internal view returns
(uint256 txCost) {
    return UpkeepLibrary.calculateUpkeepTxCost(tx.gasprice, gasOverhead,
chainlinkUpkeepTxPremiumFactor);
}

```

```

function calculateTxCostWithOverheadWithPremium(
    uint256 totalVaultHarvestOverhead
) internal view returns (uint256 txCost) {
    return
        UpkeepLibrary.calculateUpkeepTxCostFromTotalVaultHarvestOverhead(
            tx.gasprice,
            totalVaultHarvestOverhead,
            keeperRegistryGasOverhead,
            chainlinkUpkeepTxPremiumFactor
        );
}

function calculateExpectedTotalUpkeepTxCost(
    uint256 numberOfVaultsToHarvest
) internal view returns (uint256 txCost) {
    uint256 totalVaultHarvestGasOverhead = vaultHarvestFunctionGasOverhead
numberOfVaultsToHarvest;
    return
        UpkeepLibrary.calculateUpkeepTxCostFromTotalVaultHarvestOverhead(
            tx.gasprice,
            totalVaultHarvestGasOverhead,
            keeperRegistryGasOverhead,
            chainlinkUpkeepTxPremiumFactor
        );
}

function estimateSingleVaultHarvestGasOverhead(
    uint256 vaultHarvestFunctionGasOverhead
) internal view returns (uint256 totalGasOverhead) {
    totalGasOverhead = vaultHarvestFunctionGasOverhead +
keeperRegistryGasOverhead;
}

/
/ Misc /
/

/
    @dev Rescues random funds stuck.
    @param token address of the token to rescue.
/
function inCaseTokensGetStuck(address token) external onlyOwner {
    IERC20 token = IERC20(token);

    uint256 amount = token.balanceOf(address(this));
    token.safeTransfer(msg.sender, amount);
}
}

```

This is an abstract contract that iterates vaults from a provided list and fits vaults within upkeep gas limit. The contract also provides helper functions to calculate gas consumption and estimate profit and contains a trigger mechanism can be time-based, profit-based or custom. Finally, the cotract reports profits, successful harvests, and failed harvests.

getting-started.mdx:

```

---
section: chainlinkFunctions
date: Last Modified
title: "Getting Started"

```

```

metadata:
  linkToWallet: true
whatsnext:
  {
    "Try out the Chainlink Functions Tutorials":
"/chainlink-functions/tutorials",
    "Read the Architecture to understand how Chainlink Functions operates":
"/chainlink-functions/resources/architecture",
  }
---
```

```

import { Aside, CopyText, CodeSample, ClickToZoom } from "@components"
import { Tabs } from "@components/Tabs"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
import { YouTube } from "@astro-community/astro-embed-youtube"
```

Learn how to make requests to the Chainlink Functions Decentralized Oracle Network (DON) and make any computation or API calls offchain. Chainlink Functions is available on several blockchains (see the supported networks page), but this guide uses Sepolia to simplify access to testnet funds. Complete the following tasks to get started with Chainlink Functions:

- Set up your web3 wallet and fund it with testnet tokens.
- Simulate a Chainlink Functions on the Chainlink Functions Playground.
- Send a Chainlink Functions request to the DON. The JavaScript source code makes an API call to the Star Wars API and fetches the name of a given character.
- Receive the response from Chainlink Functions and parse the result.

```
<YouTube id="https://www.youtube.com/watch?v=p3CxiGwythM" />
```

Simulation

Before making a Chainlink Functions request from your smart contract, it is always a good practice to simulate the source code offchain to make any adjustments or corrections.

1. Open the Functions playground.
1. Copy and paste the following source code into the playground's code block.

```
<CodeSample lang="javascript" src="samples/ChainlinkFunctions/starwars-api.js" />
```

1. Under Argument, set the first argument to `<CopyText text="1" code/>`. You are going to fetch the name of the first Star Wars character.
1. Click on Run code. Under Output, you should see Luke Skywalker.

```
<ClickToZoom src="/images/chainlink-functions/getting-started/simulation.jpg" />
```

Configure your resources

Configure your wallet

You will test on Sepolia, so you must have an Ethereum web3 wallet with enough testnet ETH and LINK tokens. Testnet ETH is the native gas fee token on Sepolia. You will use testnet ETH tokens to pay for gas whenever you make a transaction on Sepolia. On the other hand, you will use LINK tokens to pay the Chainlink Functions Decentralized Oracles Network (DON) for processing your request.

1. Install the MetaMask wallet or other Ethereum web3 wallet.

1. Set the network for your wallet to the Sepolia testnet. If you need to add Sepolia to your wallet, you can find the chain ID and the LINK token contract address on the LINK Token Contracts page.

```
- <a
  class="erc-token-address"
  id="111551110x779877A7B0D9E8603169DdbD7836e478b4624789"
  href="/resources/link-token-contracts#sepolia-testnet"
>
  Sepolia testnet and LINK token contract
</a>
```

1. Request testnet LINK and ETH from faucets.chain.link/sepolia.

Deploy a Functions consumer contract on Sepolia

1. Open the `GettingStartedFunctionsConsumer.sol` contract in Remix.

```
<CodeSample
src="samples/ChainlinkFunctions/GettingStartedFunctionsConsumer.sol"
showButtonOnly={true} />
```

1. Compile the contract.

1. Open MetaMask and select the Sepolia network.

1. In Remix under the Deploy & Run Transactions tab, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Sepolia.

```
<ClickToZoom src="/images/chainlink-functions/getting-started/injected-
provider.jpg" />
```

1. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Sepolia.

```
<ClickToZoom src="/images/chainlink-functions/getting-started/deploy.jpg" />
```

1. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy the contract address and save it for later. You will use this address with a Functions Subscription.

```
<ClickToZoom src="/images/chainlink-functions/getting-started/deployed-
contracts.jpg" />
```

Create a subscription

You use a Chainlink Functions subscription to pay for, manage, and track Functions requests.

1. Go to functions.chain.link.

1. Click Connect wallet:

```
<ClickToZoom
src="/images/chainlink-functions/tutorials/subscription/frontend-
landing.jpg"
alt="Chainlink Functions subscription landing page"
style="max-width: 70%;"
/>
```

1. Read and accept the Chainlink Foundation Terms of Service. Then click MetaMask.

```
<ClickToZoom
```

```
src="/images/chainlink-functions/tutorials/subscription/accept-CL-  
foundation-tos.jpg"  
alt="Chainlink Functions accept ToS"  
style="max-width: 70%;"  
</>
```

1. Make sure your wallet is connected to the Sepolia testnet. If not, click the network name in the top right corner of the page and select Sepolia.

```
<ClickToZoom  
src="/images/chainlink-functions/tutorials/subscription/wallet-connected-  
ethereum-sepolia.webp"  
style="max-width: 70%;"  
alt="Chainlink Functions subscription landing page"  
</>
```

1. Click Create Subscription:

```
<ClickToZoom  
src="/images/chainlink-functions/tutorials/subscription/frontend-landing-  
wallet-connected.jpg"  
alt="Chainlink Functions subscription landing page"  
style="max-width: 70%;"  
</>
```

1. Provide an email address and an optional subscription name:

```
<ClickToZoom  
src="/images/chainlink-functions/tutorials/subscription/create-  
subscription.jpg"  
alt="Chainlink Functions create subscription"  
style="max-width: 70%;"  
</>
```

1. The first time you interact with the Subscription Manager using your EOA, you must accept the Terms of Service (ToS). A MetaMask popup appears and you are asked to accept the ToS:

```
<ClickToZoom  
src="/images/chainlink-functions/tutorials/subscription/acceptTos.jpg"  
alt="Chainlink Functions accept ToS"  
style="max-width: 70%;"  
</>
```

1. After you approve the ToS, another MetaMask popup appears, and you are asked to approve the subscription creation:

```
<ClickToZoom  
src="/images/chainlink-functions/tutorials/subscription/approve-create-  
subscription.jpg"  
alt="Chainlink Functions approve subscriptions creation"  
style="max-width: 70%;"  
</>
```

1. After the subscription is created, MetaMask prompts you to sign a message that links the subscription name and email address to your subscription:

```
<ClickToZoom  
src="/images/chainlink-functions/tutorials/subscription/sign-message-  
tos.jpg"  
alt="Chainlink Functions sign message ToS"  
style="max-width: 70%;"  
</>
```

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/sign-message-tos-
metamask.jpg"
  alt="Chainlink Functions sign message ToS MetaMask"
  style="max-width: 70%;"
/>
```

Fund your subscription

1. After the subscription is created, the Functions UI prompts you to fund your subscription. Click Add funds:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
created.jpg"
  alt="Chainlink Functions subscription created"
  style="max-width: 70%;"
/>
```

1. For this example, add 2 LINK and click Add funds:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
created-add-funds.jpg"
  alt="Chainlink Functions subscription add funds"
  style="max-width: 70%;"
/>
```

Add a consumer to your subscription

1. After you fund your subscription, add your consumer to it. Specify the address for the consumer contract that you deployed earlier and click Add consumer. MetaMask prompts you to confirm the transaction.

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
created-add-consumer.jpg"
  alt="Chainlink Functions subscription add consumer"
  style="max-width: 70%;"
/>
```

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
created-add-consumer-2.jpg"
  alt="Chainlink Functions subscription add consumer"
  style="max-width: 70%;"
/>
```

1. Subscription creation and configuration is complete. You can always see the details of your subscription again at functions.chain.link:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
details-after-creation.jpg"
  alt="Chainlink Functions subscription details"
  style="max-width: 70%;"
/>
```

Run the example

The example is hardcoded to communicate with Chainlink Functions on Sepolia. After this example is run, you can examine the code and see a detailed description of all components.

1. In Remix under the Deploy & Run Transactions tab, expand your contract in the Deployed Contracts section.
1. Expand the sendRequest function to display its parameters.
1. Fill in the subscriptionId with your subscription ID and args with [1]. You can find your subscription ID on the Chainlink Functions Subscription Manager at functions.chain.link. The [1] value for args specifies which argument in the response will be retrieved.

1. Click the transact button.

```
<ClickToZoom src="/images/chainlink-functions/getting-started/send-request.jpg" />
```

1. Wait for the request to be fulfilled. You can monitor the status of your request on the Chainlink Functions Subscription Manager.

```
<ClickToZoom src="/images/chainlink-functions/getting-started/request-fulfilled.png" />
```

1. Refresh the Functions UI to get the latest request status.

1. After the status is Success, check the character name. In Remix, under the Deploy & Run Transactions tab, click the character function. If the transaction and request ran correctly, you will see the name of your character in the response.

```
<ClickToZoom src="/images/chainlink-functions/getting-started/character.jpg" />
```

Chainlink Functions is capable of much more than just retrieving data. Try one of the Tutorials to see examples that can GET and POST to public APIs, securely handle API secrets, handle custom responses, and query multiple APIs.

Examine the code

Solidity code

```
<CodeSample src="samples/ChainlinkFunctions/GettingStartedFunctionsConsumer.sol" />
```

- To write a Chainlink Functions consumer contract, your contract must import FunctionsClient.sol and FunctionsRequest.sol. You can read the API references: FunctionsClient and FunctionsRequest.

These contracts are available in an NPM package so that you can import them from within your project.

```
import {FunctionsClient} from
"@chainlink/contracts/src/v0.8/functions/v100/FunctionsClient.sol";
import {FunctionsRequest} from
"@chainlink/contracts/src/v0.8/functions/v100/libraries/FunctionsRequest.sol";
```

- Use the FunctionsRequest.sol library to get all the functions needed for building a Chainlink Functions request.

```
using FunctionsRequest for FunctionsRequest.Request;
```

- The latest request ID, latest received response, and latest received error (if any) are defined as state variables:


```
bytes32 public slastRequestId;  
bytes public slastResponse;  
bytes public slastError;
```

- We define the Response event that your smart contract will emit during the callback

```
event Response(bytes32 indexed requestId, string character, bytes response,  
bytes err);
```

- The Chainlink Functions router address and donID are hardcoded for Sepolia. Check the supported networks page to try the code sample on another testnet.

- The gasLimit is hardcoded to 3000000, the amount of gas that Chainlink Functions will use to fulfill your request.

- The JavaScript source code is hardcoded in the source state variable. For more explanation, read the JavaScript code section.

- Pass the router address for your network when you deploy the contract:

```
constructor() FunctionsClient(router)
```

- The two remaining functions are:

- sendRequest for sending a request. It receives the subscription ID and list of arguments to pass to the source code. Then:

- It uses the FunctionsRequest library to initialize the request and add the source code and arguments. You can read the API Reference for Initializing a request and adding arguments.

```
FunctionsRequest.Request memory req;  
req.initializeRequestForInlineJavaScript(source);  
if (args.length > 0) req.setArgs(args);
```

- It sends the request to the router by calling the FunctionsClient sendRequest function. You can read the API reference for sending a request. Finally, it stores the request id in slastRequestId and returns it.

```
slastRequestId = sendRequest(  
    req.encodeCBOR(),  
    subscriptionId,  
    gasLimit,  
    jobId  
);  
return slastRequestId;
```

Note: sendRequest accepts requests encoded in bytes. Therefore, you must encode it using encodeCBOR.

- fulfillRequest to be invoked during the callback. This function is defined in FunctionsClient as virtual (read fulfillRequest API reference). So, your smart contract must override the function to implement the callback. The

implementation of the callback is straightforward: the contract stores the latest response and error in `slastResponse` and `slastError`, parses the response from bytes to string to fetch the character name before emitting the `Response` event.

```
slastResponse = response;
character = string(response);
slastError = err;
emit Response(requestId, slastResponse, slastError);
```

JavaScript code

```
<CodeSample lang="javascript" src="samples/ChainlinkFunctions/starwars-api.js" />
```

This JavaScript source code uses `Functions.makeHttpRequest` to make HTTP requests. The source code calls the `https://swapi.info/` API to request a Star Wars character name. If you read the `Functions.makeHttpRequest` documentation and the Star Wars API documentation, you notice that URL has the following format where `$characterId` is provided as parameter when making the HTTP request:

url: `https://swapi.info/api/people/${characterId}/`

To check the expected API response for the first character, you can directly paste the following URL in your browser `https://swapi.info/api/people/1/` or run the `curl` command in your terminal:

```
bash
curl -X 'GET' \
  'https://swapi.info/api/people/1/' \
  -H 'accept: application/json'
```

The response should be similar to the following example:

```
json
{
  "name": "Luke Skywalker",
  "height": "172",
  "mass": "77",
  "haircolor": "blond",
  "skincolor": "fair",
  "eyecolor": "blue",
  "birthyear": "19BBY",
  "gender": "male",
  "homeworld": "https://swapi.info/api/planets/1/",
  "films": [
    "https://swapi.info/api/films/1/",
    "https://swapi.info/api/films/2/",
    "https://swapi.info/api/films/3/",
    "https://swapi.info/api/films/6/"
  ],
  "species": [],
  "vehicles": ["https://swapi.info/api/vehicles/14/",
"https://swapi.info/api/vehicles/30/"],
  "starships": ["https://swapi.info/api/starships/12/",
"https://swapi.info/api/starships/22/"],
  "created": "2014-12-09T13:50:51.644000Z",
  "edited": "2014-12-20T21:17:56.891000Z",
  "url": "https://swapi.info/api/people/1/"
}
```

```
}
```

Now that you understand the structure of the API. Let's delve into the JavaScript code. The main steps are:

- Fetch `characterId` from `args`. `Args` is an array. The `characterId` is located in the first element.
- Make the HTTP call using `Functions.makeHttpRequest` and store the response in `apiResponse`.
- Throw an error if the call is not successful.
- The API response is located at `data`.
- Read the name from the API response `data.name` and return the result as a buffer using the `Functions.encodeString` helper function. Because the name is a string, we use `encodeString`. For other data types, you can use different data encoding functions.

Note: Read this article if you are new to Javascript Buffers and want to understand why they are important.

```
# index.mdx:
```

```
---
section: chainlinkFunctions
date: Last Modified
title: "Chainlink Functions Beta"
isIndex: true
whatsnext:
  {
    "Learn the basics in the Getting Started guide.":
"/chainlink-functions/getting-started",
    "Learn how to use more advanced capabilities in one of the Tutorials.":
"/chainlink-functions/tutorials",
    "Learn about core concepts, the Chainlink Functions architecture, and
billing.": "/chainlink-functions/resources",
  }
---
```

```
import { Aside } from "@components"
```

```
<Aside type="note" title="Mainnet Beta">
Chainlink Functions is available on mainnet only as a BETA preview to ensure
that this new platform is robust and secure for developers. While in BETA,
developers must follow best practices and not use the BETA for any mission-
critical application or secure any value. Chainlink Functions is likely to
evolve and improve. Breaking changes might occur while the service is in BETA.
Monitor these docs to stay updated on feature improvements along with interface
and contract changes.
```

```
</Aside>
```

Chainlink Functions provides your smart contracts access to trust-minimized compute infrastructure, allowing you to fetch data from APIs and perform custom computation. Your smart contract sends source code in a request to a Decentralized Oracle Network (DON), and each node in the DON executes the code in a serverless environment. The DON then aggregates all the independent return values from each execution and sends the final result back to your smart contract.

Chainlink Functions eliminates the need for you to manage your own Chainlink node and provides decentralized offchain computation and consensus, ensuring that a minority of the network cannot manipulate the response sent back to your smart contract.

Furthermore, Chainlink Functions allows you to include secret values in your request that are encrypted using threshold encryption. These values can only be decrypted via a multi-party decryption process, meaning that every node can only decrypt the secrets with participation from other DON nodes. This feature can provide API keys or other sensitive values to your source code, enabling access to APIs that require authentication.

To pay for requests, you fund a subscription account with LINK. Your subscription is billed when the DON fulfills your requests. Check out the [subscriptions page](#) for more information.

Read the [architecture page](#) to learn more about how Chainlink Functions works.

See the [Tutorials page](#) for simple tutorials showing you different GET and POST requests that run on Chainlink Functions. You can also gain hands-on experience with Chainlink Functions with the [Chainlink Functions Playground](#).

When to use Chainlink Functions

<Aside type="note">

Chainlink Functions is a self-service solution. You are responsible for independently reviewing any code and API dependencies that you submit in a request. Community-created code examples might not be audited, so you must independently review this code before you use it.

Chainlink Functions is offered "as is" and "as available" without conditions or warranties of any kind. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs from Functions due to issues in your code or downstream issues with API dependencies. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. Users are responsible for complying with the licensing agreements for all data providers that they connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account or subscription.

</Aside>

Chainlink Functions enables a variety of use cases. Use Chainlink Functions to:

- Connect to any public data. For example, you can connect your smart contracts to weather statistics for parametric insurance or real-time sports results for Dynamic NFTs.
- Connect to public data and transform it before consumption. You could calculate Twitter sentiment after reading data from the Twitter API, or derive asset prices after reading price data from Chainlink Price Feeds.
- Connect to a password-protected data source; from IoT devices like smartwatches to enterprise resource planning systems.
- Connect to an external decentralized database, such as IPFS, to facilitate offchain processes for a dApp or build a low-cost governance voting system.
- Connect to your Web2 application and build complex hybrid smart contracts.
- Fetch data from almost any Web2 system such as AWS S3, Firebase, or Google Cloud Storage.

You can find several community examples at [useChainlinkFunctions.com](https://usechainlinkfunctions.com)

Supported networks

See the [Supported Networks page](#) to find a list of supported networks and contract addresses.

service-responsibility.mdx:

```

---
section: chainlinkFunctions
date: Last Modified
title: "Chainlink Functions Service Responsibility"
metadata:
  linkToWallet: true
whatsnext:
  {
    "Try out the Chainlink Functions Tutorials":
"/chainlink-functions/tutorials",
    "Read the Architecture to understand how Chainlink Functions operates":
"/chainlink-functions/resources/architecture",
  }
---

```

```
import { Aside } from "@components"
```

Chainlink Functions provides access to trust-minimized compute infrastructure that allows you to retrieve data and run computation. Because the service is highly-flexible and runs offchain, both developers and Chainlink service providers share responsibility in ensuring that operation and performance match expectations.

Developer responsibilities

Developers who implement Chainlink Functions in their code and applications are responsible for several components.

Data

- Data quality: When using Chainlink Functions, developers are responsible for ensuring that any external data they retrieve meet the data quality requirements for their applications. This responsibility applies to data retrieved via APIs or other data retrieval mechanisms. You must ensure that the data sources you consume through Chainlink Functions are accurate, reliable, secure, and not at risk of manipulation by malicious actors. When possible, use multiple data sources to reduce single points of failure and manipulation risks.
- Data availability: Developers must ensure that the data sources and APIs that they use with Chainlink Functions meet the fault-tolerance and availability requirements for their applications. Node operators are geographically distributed, so developers must ensure that APIs do not have geographic or other restrictions that would prohibit node operators from retrieving data. When possible, use redundant data sources to reduce the risk that your applications cannot execute due to unavailable data. In situations where data isn't available, ensure proper error handling.
- Data privacy and ethics: Developers are responsible for determining what data they have a right to use with Chainlink Functions and ensuring that they use that data ethically. Developers must ensure that their Chainlink Functions code does not expose their private information or the private information of users without proper consent. Do not use data that you are not authorized to access. Your Chainlink Functions code and your application must ensure that private or sensitive information remains secure.

Code

- Code quality and reliability: Developers must execute code on Chainlink Functions only if the code meets the quality and reliability requirements for their use case and application.
- Code and application audits: Developers are responsible for auditing their code and applications before deploying to production. You must determine the quality of any audits and ensure that they meet the requirements for your application and any code that runs on Chainlink Functions.
- Code dependencies and imports: Developers are responsible for ensuring the quality, reliability, and security of any dependencies or imported packages that

they use with Chainlink Functions. Review and audit these dependencies and packages.

Secrets

<Aside type="caution">

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

</Aside>

- Self-hosted secrets: Developers are responsible for securing self-hosted secrets, monitoring unauthorized access, auditing permissions, and ensuring that secrets are available for retrieval by Chainlink Functions when executing code.
- Secrets best practices: For all types of secrets used with Chainlink Functions, developers must follow common best practices for managing secrets for applications. Developers are responsible for selecting, setting expiration time, monitoring, and rotating secrets to ensure the security of their applications and Chainlink Functions code.

Subscriptions

- Subscription owner wallet management: Developers must ensure the security of any wallets that own Chainlink Functions subscriptions or wallets that secure funds for subscriptions.
- Subscription balances: Subscription owners are responsible for maintaining the Chainlink Function balance that is necessary to fund Chainlink Functions requests and computation. Monitor your subscription balance and implement the necessary processes to fund your subscription balance at a level that meets your application's requirements.

Node Operator responsibilities

High-quality node operators participate in the Functions DONs using a configuration specified in the Chainlink software. As participants in these deployments, Node Operators are responsible for the following components of Chainlink Functions:

- Ensuring the proper configuration, maintenance, and monitoring of nodes participating in the Chainlink Functions DON.
- Storing encrypted secrets that developers provide using threshold encryption.
- Ensuring that transactions execute onchain in a timely manner and apply gas bumping when necessary.
- Selecting and properly employing blockchain clients to connect to supported blockchain networks.
- Maintaining continuous uptime and active participation in OCR consensus.
- Employ defensive measures to prevent unauthorized access to their Chainlink node deployments.
- Ensure that Chainlink node deployments are running the latest software versions.
- Responding to important communication from Chainlink Labs or from other node operators in a timely manner.

supported-networks.mdx:

section: chainlinkFunctions
date: Last Modified
title: "Supported Networks"

```
import ResourcesCallout from
"@features/resources/callouts/ResourcesCallout.astro"
import { Aside, Address, CopyText } from "@components"
import { DonId } from "@features/chainlink-functions"
```

Read the LINK Token Contracts page to learn where to get testnet LINK and ETH.

```
<ResourcesCallout callout="bridgeRisks" />
```

Ethereum

```
<Aside type="note" title="USD-denominated premium fees">
```

The fixed premium fee is denominated in USD but no USD is ever used. The LINK equivalent of the fee is calculated at

request time, and then deducted from your subscription in LINK at response time. See the example cost calculation for more information.

```
</Aside>
```

Ethereum mainnet

Item	Value
Functions router	<code><Address contractUrl="https://etherscan.io/address/0x65Dcc24F8ff9e51F10DCc7Ed1e4e2A61e6E14bd6" /></code>
Encrypted secrets upload endpoints	<code><CopyText text="https://01.functions-gateway.chain.link/" code /><CopyText text="https://02.functions-gateway.chain.link/" code /></code>

Billing Item	Value
Premium fees	400 cents USD
Request threshold (withdrawing funds)	1 request
Cancellation fees (withdrawing funds)	1 LINK
Minimum balance for uploading encrypted secrets	1 LINK

Sepolia testnet

Item	Value
Functions router	<code><Address contractUrl="https://sepolia.etherscan.io/address/0xb83E47C2bC239B3bf370bc41e1459A34b41238D0" /></code>
Encrypted secrets upload endpoints	<code><CopyText text="https://01.functions-gateway.chain.link/" code /><CopyText text="https://02.functions-gateway.chain.link/" code /></code>

```
text="https://01.functions-gateway.testnet.chain.link/" code
/></li><li><CopyText text="https://02.functions-gateway.testnet.chain.link/"
code /></li></ul> |
```

Billing Item	Value
Premium fees	320 cents USD
Request threshold (withdrawing funds)	10 requests
Cancellation fees (withdrawing funds)	2 LINK
Minimum balance for uploading encrypted secrets	2 LINK

Polygon

<Aside type="note" title="USD-denominated premium fees">

The fixed premium fee is denominated in USD but no USD is ever used. The LINK equivalent of the fee is calculated at

request time, and then deducted from your subscription in LINK at response time. See the example cost

calculation for more information.

</Aside>

Polygon mainnet

Item	Value
-----	-----
-----	-----
-----	-----
Functions router	<Address contractUrl="https://polygonscan.com/address/0xdc2AAF042Aeff2E68B3e8E33F19e4B9fA7C73F10" />
<DonId client:load/>	<CopyText text="fun-polygon-mainnet-1"
code /> / <CopyText	
text="0x66756e2d706f6c79676f6e2d6d61696e6e65742d310000000000000000000000"code	
format formatType="bytes32" />	
Encrypted secrets upload endpoints	<CopyText
text="https://01.functions-gateway.chain.link/" code /><CopyText	
text="https://02.functions-gateway.chain.link/" code />	
-----	-----
Billing Item	Value
-----	-----
Premium fees	3 cents USD
Request threshold (withdrawing funds)	1 request
Cancellation fees (withdrawing funds)	1 LINK
Minimum balance for uploading encrypted secrets	1 LINK

Polygon Amoy testnet

Item	Value
-----	-----
-----	-----
-----	-----
Functions router	<Address contractUrl="https://amoy.polygonscan.com/address/0xC22a79eBA640940ABB6dF0f7982cc119578E11De" />
<DonId client:load/>	<CopyText text="fun-polygon-amoy-1"
code /> / <CopyText	
text="0x66756e2d706f6c79676f6e2d616d6f792d3100000000000000000000000000"code	
format formatType="bytes32" />	
Encrypted secrets upload endpoints	<CopyText


```
text="https://01.functions-gateway.testnet.chain.link/" code
/></li><li><CopyText text="https://02.functions-gateway.testnet.chain.link/"
code /></li></ul> |
```

Billing Item	Value
Premium fees (converted to LINK at request time)	320 cents USD
Request threshold (withdrawing funds)	10 requests
Cancellation fees (withdrawing funds)	2 LINK
Minimum balance for uploading encrypted secrets	2 LINK

Avalanche

<Aside type="note" title="USD-denominated premium fees">

The fixed premium fee is denominated in USD but no USD is ever used. The LINK equivalent of the fee is calculated at

request time, and then deducted from your subscription in LINK at response time. See the example cost

calculation for more information.

</Aside>

Avalanche Mainnet

```
| Item | Value
|-----|
| Functions router | <Address
contractUrl="https://snowtrace.io/address/0x9f82a6A0758517FD0AfA463820F586999AF3
14a0" />
| <DonId client:load/> | <CopyText text="fun-avalanche-mainnet-1"
code /> / <CopyText
text="0x66756e2d6176616c616e6368652d6d61696e6e65742d31000000000000000000"code
format formatType="bytes32" /> |
| Encrypted secrets upload endpoints | <ul><li><CopyText
text="https://01.functions-gateway.chain.link/" code /></li><li><CopyText
text="https://02.functions-gateway.chain.link/" code /></li></ul> |
```

Billing Item	Value
Premium fees	3 cents USD
Request threshold (withdrawing funds)	1 request
Cancellation fees (withdrawing funds)	1 LINK
Minimum balance for uploading encrypted secrets	1 LINK

Avalanche Fuji testnet

```
| Item | Value
|-----|
| Functions router | <Address
contractUrl="https://testnet.snowtrace.io/address/0xA9d587a00A31A52Ed70D6026794a
8FC5E2F5dCb0" />
| <DonId client:load/> | <CopyText text="fun-avalanche-fuji-1"
code /> / <CopyText
text="0x66756e2d6176616c616e6368652d66756a692d31000000000000000000000000"code
format formatType="bytes32" /> |
```

| Encrypted secrets upload endpoints |

- `<CopyText text="https://01.functions-gateway.testnet.chain.link/" code /><CopyText text="https://02.functions-gateway.testnet.chain.link/" code /> |`

Billing Item	Value
Premium fees	320 cents USD
Request threshold (withdrawing funds)	10 requests
Cancellation fees (withdrawing funds)	2 LINK
Minimum balance for uploading encrypted secrets	2 LINK

Arbitrum

`<Aside type="note" title="USD-denominated premium fees">`

The fixed premium fee is denominated in USD but no USD is ever used. The LINK equivalent of the fee is calculated at request time, and then deducted from your subscription in LINK at response time. See the example cost calculation for more information.

`</Aside>`

Arbitrum Mainnet

Item	Value

Functions router	<code><Address contractUrl="https://arbiscan.io/address/0x97083e831f8f0638855e2a515c90edcf158df238" /></code>
<code><DonId client:load/></code>	<code><CopyText text="fun-arbitrum-mainnet-1" code /> / <CopyText text="0x66756e2d617262697472756d2d6d61696e6e65742d3100000000000000000000"code format formatType="bytes32" /> </code>
Encrypted secrets upload endpoints	<code><CopyText text="https://01.functions-gateway.chain.link/" code /><CopyText text="https://02.functions-gateway.chain.link/" code /> </code>
Billing Item	Value
-----	-----
Premium fees	3 cents USD
Request threshold (withdrawing funds)	10 requests
Cancellation fees (withdrawing funds)	0.1 LINK
Minimum balance for uploading encrypted secrets	0.1 LINK

Arbitrum Sepolia testnet

Item	Value

Functions router	<code><Address contractUrl="https://sepolia.arbiscan.io/address/0x234a5fb5Bd614a7AA2FfAB244D603abFA0Ac5C5C" /></code>
<code><DonId client:load/></code>	<code><CopyText text="fun-arbitrum-sepolia-1" code /> / <CopyText text="0x66756e2d617262697472756d2d7365706f6c69612d3100000000000000000000"code</code>

```
format formatType="bytes32" /> |
| Encrypted secrets upload endpoints | <ul><li><CopyText
text="https://01.functions-gateway.testnet.chain.link/" code
/></li><li><CopyText text="https://02.functions-gateway.testnet.chain.link/"
code /></li></ul> |
```

Billing Item	Value
Premium fees	320 cents USD
Request threshold (withdrawing funds)	10 requests
Cancellation fees (withdrawing funds)	2 LINK
Minimum balance for uploading encrypted secrets	2 LINK

BASE

<Aside type="note" title="USD-denominated premium fees">
The fixed premium fee is denominated in USD but no USD is ever used. The LINK equivalent of the fee is calculated at request time, and then deducted from your subscription in LINK at response time. See the example cost calculation for more information.
</Aside>

BASE Mainnet

```
| Item | Value |
| ----- |
| Functions router | <Address
contractUrl="https://basescan.org/address/0xf9b8fc078197181c841c296c876945aaa425b278" />
| <DonId client:load/> | <CopyText text="fun-base-mainnet-1"
code /> / <CopyText
text="0x66756e2d626173652d6d61696e6e65742d31000000000000000000000000000000"code
format formatType="bytes32" /> |
| Encrypted secrets upload endpoints | <ul><li><CopyText
text="https://01.functions-gateway.chain.link/" code /></li><li><CopyText
text="https://02.functions-gateway.chain.link/" code /></li></ul> |
```

Billing Item	Value
Premium fees (converted to LINK at request time)	3 cents USD
Request threshold (withdrawing funds)	1 request
Cancellation fees (withdrawing funds)	0.1 LINK
Minimum balance for uploading encrypted secrets	0.1 LINK

BASE Sepolia testnet

```
| Item | Value |
| ----- |
| Functions router | <Address
contractUrl="https://sepolia.basescan.org/address/0xf9B8fc078197181C841c296C876945aaa425B278" />
| <DonId client:load/> | <CopyText text="fun-base-sepolia-1"
code /> / <CopyText
text="0x66756e2d626173652d7365706f6c69612d31000000000000000000000000000000"code
```

```
format formatType="bytes32" /> |
| Encrypted secrets upload endpoints | <ul><li><CopyText
text="https://01.functions-gateway.testnet.chain.link/" code
/></li><li><CopyText text="https://02.functions-gateway.testnet.chain.link/"
code /></li></ul> |
```

Billing Item	Value
Premium fees (converted to LINK at request time)	320 cents USD
Request threshold (withdrawing funds)	10 requests
Cancellation fees (withdrawing funds)	2 LINK
Minimum balance for uploading encrypted secrets	2 LINK

Optimism

<Aside type="note" title="USD-denominated premium fees">
 The fixed premium fee is denominated in USD but no USD is ever used. The LINK equivalent of the fee is calculated at request time, and then deducted from your subscription in LINK at response time. See the example cost calculation for more information.
 </Aside>

Optimism Sepolia testnet

Item	Value
Functions router	<Address contractUrl="https://sepolia-optimism.etherscan.io/address/0xC17094E3A1348E5C7544D4fF8A36c28f2C6AAE28" />
<DonId client:load/>	<CopyText text="fun-optimism-sepolia-1" code /> / <CopyText text="0x66756e2d6f7074696d69736d2d7365706f6c69612d3100000000000000000000" code format formatType="bytes32" />
Encrypted secrets upload endpoints	<CopyText text="https://01.functions-gateway.testnet.chain.link/" code /><CopyText text="https://02.functions-gateway.testnet.chain.link/" code />

Billing Item	Value
Premium fees (converted to LINK at request time)	320 cents USD
Request threshold (withdrawing funds)	10 requests
Cancellation fees (withdrawing funds)	2 LINK
Minimum balance for uploading encrypted secrets	2 LINK

functions-client.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "FunctionsClient API Reference"
---
```

```
import { Aside, CopyText } from "@components"
```

<Aside type="note" title="Add Chainlink to your project">
 If you need to integrate Chainlink into your project, install the @chainlink/contracts NPM package.

- If you use NPM: <CopyText text="npm install @chainlink/contracts --save" code/>

- If you use Yarn: <CopyText text="yarn add @chainlink/contracts" code/>

Functions contracts are available starting from version 0.7.1.

</Aside>

Consumer contract developers inherit FunctionsClient to create Chainlink Functions requests.

Events

RequestSent

```
solidity
event RequestSent(bytes32 id)
```

RequestFulfilled

```
solidity
event RequestFulfilled(bytes32 id)
```

Errors

OnlyRouterCanFulfill

```
solidity
error OnlyRouterCanFulfill()
```

Methods

constructor

```
solidity
constructor(address router)
```

\sendRequest

```
solidity
function sendRequest(bytes data, uint64 subscriptionId, uint32 callbackGasLimit,
bytes32 donId) internal returns (bytes32)
```

Sends a Chainlink Functions request to the stored router address

Parameters

Name	Type	Description
data	bytes	The CBOR encoded bytes data for a Functions request
subscriptionId	uint64	The subscription ID that will be charged to service the request
callbackGasLimit	uint32	the amount of gas that will be available for the fulfillment callback

```
| donId          | bytes32 |
|
```

Return Values

Name	Type	Description
[0]	bytes32	requestId The generated request ID for this request

fulfillRequest

solidity

```
function fulfillRequest(bytes32 requestId, bytes response, bytes err) internal
virtual
```

User defined function to handle a response from the DON

Either response or error parameter will be set, but never both

Parameters

Name	Type	Description
requestId	bytes32	The request ID, returned by sendRequest()
response	bytes	Aggregated response from the execution of the user's source code
err	bytes	Aggregated error from the execution of the user code or from the execution pipeline

handleOracleFulfillment

solidity

```
function handleOracleFulfillment(bytes32 requestId, bytes response, bytes err)
external
```

Chainlink Functions response handler called by the Functions Router during fulfillment from the designated transmitter node in an OCR round

Either response or error parameter will be set, but never both

Parameters

Name	Type	Description
requestId	bytes32	The requestId returned by FunctionsClient.sendRequest().
response	bytes	Aggregated response from the request's source code.
err	bytes	Aggregated error either from the request's source code or from the execution pipeline.

functions-request.mdx:

```
section: chainlinkFunctions
date: Last Modified
title: "FunctionsRequest library API Reference"
---
```

```
import { Aside, CopyText } from "@components"
```

```
<Aside type="note" title="Add Chainlink to your project">
```

```
  If you need to integrate Chainlink into your project, install the
  @chainlink/contracts NPM package.
```

```
- If you use NPM: <CopyText text="npm install @chainlink/contracts --save"
code/>
```

```
- If you use Yarn: <CopyText text="yarn add @chainlink/contracts" code/>
```

```
Functions contracts are available starting from version 0.7.1.
```

```
</Aside>
```

Consumer contract developers use the FunctionsRequest library to build their requests.

Types and Constants

REQUESTDATAVERSION

```
solidity
uint16 REQUESTDATAVERSION
```

DEFAULTBUFFERSIZE

```
solidity
uint256 DEFAULTBUFFERSIZE
```

Location

```
solidity
enum Location {
  Inline,
  Remote,
  DONHosted
}
```

Value	Description

Inline	Provided within the Request.
Remote	Hosted through a remote location that can be accessed through a provided URL.
DONHosted	Hosted on the DON's storage.

CodeLanguage

```
solidity
enum CodeLanguage {
  JavaScript
}
```

Request

```
solidity
struct Request {
    enum FunctionsRequest.Location codeLocation;
    enum FunctionsRequest.Location secretsLocation;
    enum FunctionsRequest.CodeLanguage language;
    string source;
    bytes encryptedSecretsReference;
    string[] args;
    bytes[] bytesArgs;
}
```

Field	Type	Description
codeLocation	Location	The location of the source code that will be executed on each node in the DON.
secretsLocation	Location	The location of secrets that will be passed into the source code. \Only Remote secrets are supported.
language	CodeLanguage	The coding language that the source code is written in.
source	string	Raw source code for Request.codeLocation of Location.Inline, URL for Request.codeLocation of Location.Remote, or slot decimal number for Request.codeLocation of Location.DONHosted.
encryptedSecretsReference	bytes	Encrypted URLs for Request.secretsLocation of Location.Remote, or CBOR encoded slotid+version for Request.secretsLocation of Location.DONHosted.
args	string[]	String arguments that will be passed into the source code.
bytesArgs	bytes[]	Bytes arguments that will be passed into the source code.

Errors

EmptySource

```
solidity
error EmptySource()
```

EmptySecrets

```
solidity
error EmptySecrets()
```

EmptyArgs

```
solidity
error EmptyArgs()
```


NoInlineSecrets

```
solidity
error NoInlineSecrets()
```

Functions

encodeCBOR

```
solidity
function encodeCBOR(struct FunctionsRequest.Request self) internal pure returns
(bytes)
```

Encodes a Request to CBOR encoded bytes

Parameters

Name	Type	Description
self	struct FunctionsRequest.Request	The request to encode

Return values

Name	Type	Description
[0]	bytes	CBOR encoded bytes

initializeRequest

```
solidity
function initializeRequest(struct FunctionsRequest.Request self, enum
FunctionsRequest.Location codeLocation, enum FunctionsRequest.CodeLanguage
language, string source) internal pure
```

Initializes a Chainlink Functions Request

Sets the codeLocation and code on the request

Parameters

Name	Type	Description
self	struct FunctionsRequest.Request	The uninitialized request
codeLocation	enum FunctionsRequest.Location	The user provided source code location
language	enum FunctionsRequest.CodeLanguage	The programming language of the user code
source	string	The user provided source code or a url

initializeRequestForInlineJavaScript

```
solidity
function initializeRequestForInlineJavaScript(struct FunctionsRequest.Request
self, string javascriptSource) internal pure
```

Initializes a Chainlink Functions Request

Simplified version of initializeRequest for PoC

Parameters

Name	Type	Description
-----	-----	
self	struct FunctionsRequest.Request	The uninitialized request
javascriptSource (must not be empty)	string	The user provided JS code

addSecretsReference

solidity

```
function addSecretsReference(struct FunctionsRequest.Request self, bytes  
encryptedSecretsReference) internal pure
```

Adds Remote user encrypted secrets to a Request

Parameters

Name	Type	Description
-----	-----	
self	struct FunctionsRequest.Request	The initialized request
encryptedSecretsReference	bytes	Encrypted comma-separated string of URLs pointing to offchain secrets

addDONHostedSecrets

solidity

```
function addDONHostedSecrets(struct FunctionsRequest.Request self, uint8 slotID,  
uint64 version) internal pure
```

Adds DON-hosted secrets reference to a Request

Parameters

Name	Type	Description
-----	-----	
self	struct FunctionsRequest.Request	The initialized request
slotID	uint8	Slot ID of the user's secrets
hosted on DON		
version	uint64	User data version (for the slotID)

setArgs

solidity

```
function setArgs(struct FunctionsRequest.Request self, string[] args) internal  
pure
```

Sets args for the user run function

Parameters

Name	Type	Description
----	-----	-----
self	struct FunctionsRequest.Request	The initialized request
args	string[]	The array of string args (must not be empty)

setBytesArgs

solidity

```
function setBytesArgs(struct FunctionsRequest.Request self, bytes[] args)
internal pure
```

Sets bytes args for the user run function

Parameters

Name	Type	Description
----	-----	-----
self	struct FunctionsRequest.Request	The initialized request
args	bytes[]	The array of bytes args (must not be empty)

javascript-source.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "JavaScript code API Reference"
---
```

JavaScript source code for a Functions request should comply with certain restrictions:

- Allowed Modules: Vanilla Deno and module imports.
- Return Type: Must return a JavaScript Buffer object representing the response bytes sent back to the invoking contract.
- Time Limit: Scripts must execute within a 10-second timeframe; otherwise, they will be terminated, and an error will be returned to the requesting contract.

HTTP requests

For making HTTP requests, use the Functions.makeHttpRequest function.

Syntax

```
javascript
const response = await Functions.makeHttpRequest({
  url: "http://example.com",
  method: "GET", // Optional
  // Other optional parameters
})
```

Parameters

Parameter Default Value	Optionality	Description	
-----	-----	-----	-----
url	Required	The target URL.	N/A
method	Optional	HTTP method to be used.	'GET'
headers	Optional	HTTP headers for the request.	N/A
params	Optional	URL query parameters.	N/A
data	Optional	Body content for the request.	N/A
timeout ms	Optional	Maximum request duration in milliseconds.	3000
responseType 'json'	Optional	Expected response type.	

Return Object

Response Type	Fields	Description
-----	-----	-----
Success	data	Response data sent by the server.
	status	Numeric HTTP status code.
	statusText	Textual representation of HTTP status.
	headers	HTTP headers sent by the server in the response.
Error	error	Indicates an error occurred (true).
	message	Optional error message.
	code	Optional error code.
	response	Optional server response.

Data encoding functions

The Functions library includes several encoding functions, which are useful for preparing data for blockchain contracts.

Function	Input Type	Output Type	Description
-----	-----	-----	-----
Functions.encodeUint256	Positive Integer	32-byte Buffer	Converts a positive integer to a 32-byte Buffer for a uint256 in Solidity.
Functions.encodeInt256	Integer	32-byte Buffer	Converts an integer to a 32-byte Buffer for an int256 in Solidity.
Functions.encodeString	String	Buffer	Converts a string to a Buffer for a string type in Solidity.

Note: Using these encoding functions is optional. The source code must return a Uint8Array which represents the bytes that are returned onchain.

```
javascript
const myArr = new Uint8Array(ARRAYLENGTH)
```

architecture.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "Chainlink Functions Architecture"
whatsnext: { "Billing": "/chainlink-functions/resources/billing" }
---
```

```
import { Aside, ClickToZoom } from "@components"
```

```
<Aside type="note" title="Prerequisites">
  Read the Chainlink Functions introduction to understand all the concepts
  discussed on this
  page.
</Aside>
```

Request and Receive Data

This model is similar to the Basic Request Model: The consumer contract initiates the cycle by sending a request to the FunctionsRouter contract. Oracle nodes watch for events emitted by the FunctionsCoordinator contract and run the computation offchain. Finally, oracle nodes use the Chainlink OCR protocol to aggregate all the returned before passing the single aggregated response back to the consumer contract via a callback function.

```
<ClickToZoom
  src="/images/chainlink-functions/requestAndReceive.png"
  alt="Chainlink Functions request and receive diagram"
/>
```

The main actors and components are:

- Initiator (or end-user): initiates the request to Chainlink Functions. It can be an EOA (Externally Owned Account) or Chainlink Automation.
- Consumer contract: smart contract deployed by developers, which purpose is to interact with the FunctionsRouter to initiate a request.
- FunctionsRouter contract: manages subscriptions and is the entry point for consumers. The interface of the router is stable. Consumers call the `sendRequest` method to initiate a request.
- FunctionsCoordinator contracts: interface for the Decentralized Oracle Network. Oracle nodes listen to events emitted by the coordinator contract and interact with the coordinator to transmit the responses.
- DON: Chainlink Functions are powered by a Decentralized Oracle Network. The oracle nodes are independent of each other and are responsible for executing the request's source code. The nodes use the Chainlink OCR protocol to aggregate all the nodes' responses. Finally, a DON's oracle sends the aggregate response to the consumer contract in a callback.
- Secrets endpoint: To transmit their secrets, users can encrypt them with the DON public key and then upload them to the secrets endpoint, a highly available service for securely sharing encrypted secrets with the nodes. Note: An alternative method involves self-hosting secrets. In this approach, users provide a publicly accessible HTTP(s) URL, allowing nodes to retrieve the encrypted secrets. Refer to the secrets management page for detailed information on both methods.
- Serverless Environment: Every Oracle node accesses a distinct, sandboxed environment for computation. While the diagram illustrates an API request, the computation isn't restricted solely to this. You can perform any computation, from API calls to mathematical operations, using vanilla Deno code without

module imports. Note: All nodes execute identical computations. If the target API has throttling limits, know that multiple simultaneous calls will occur since each DON node will independently run the request's source code.

Let's walk through the sequence of interactions among these components, as illustrated in the diagram:

1. If there are secrets, a user encrypts secrets with the public key linked to the DON master secret key (MSK) and then uploads the encrypted secrets to the secrets endpoint. The secrets endpoint pushes the encrypted secrets to the nodes part of the DON (The secrets capability depicted in this diagram is called threshold encryption and is explained in secrets management).
1. An EOA (Externally Owned Account) or Chainlink Automation initiates the request by calling the consumer contract.
1. The consumer contract should inherit the FunctionsClient contract. This ensures it will be able to receive responses from the FunctionsRouter contract via the handleOracleFulfillment callback. The router contract starts the billing to estimate the fulfillment costs and block the amount in the reservation balance (To learn more, read Cost simulation). Then it calls the FunctionsCoordinator contract.
1. The coordinator contract emits an OracleRequest event containing information about the request.
1. On reception of the event by the DON, the DON's nodes decrypt the secrets using threshold decryption (The threshold encryption feature is explained in secrets management). Each DON's Oracle node executes the request's source code in a serverless environment.
1. The DON runs the Offchain Reporting protocol (OCR) to aggregate the values returned by each node's execution of the source code.
1. A DON's oracle node transmits the attested report (which includes the aggregated response) to the FunctionsCoordinator contract.
1. The FunctionsCoordinator contract calls the FunctionsRouter's fulfill method to calculate the fulfillment costs and finalize the billing (To learn more, read Cost calculation).
1. The FunctionsRouter contract calls the consumer contract's callback with the aggregated response.

Note: Chainlink Functions requests are not limited to API requests. The diagram depicts an example of API requests, but you can request the DON to run any computation.

Subscription management

Chainlink Functions do not require your consumer contracts to hold LINK tokens and send them to oracles when making requests. Instead, you must create a subscription account and fund it to pay for your Chainlink Functions requests, so your consumer contracts don't need to hold LINK when calling Chainlink Functions.

Concepts

- Terms of service (ToS): Before interacting with Chainlink Functions, users must agree to the terms of service. Once signed, the accounts that can manage subscriptions are added to the allowedSenders in the ToS allow list contract.
- Chainlink Functions Subscription Manager: A user interface that allows users to agree to the sign Terms of service and interact with the FunctionsRouter to manage subscriptions.
- Subscription account: An account that holds LINK tokens and makes them available to fund requests to Chainlink DON. A Subscription ID uniquely identifies each account.
- Subscription ID: 64-bit unsigned integer representing the unique identifier of the Subscription account.
- Subscription owner: The wallet address that creates and manages a Subscription account. Any account can add LINK tokens to the subscription balance. Still, only the owner can add consumer contracts to the subscription, remove consumer

contracts from the subscription, withdraw funds, or transfer a subscription. Only the subscription owner can generate encrypted secrets for requests that use their Subscription ID.

- Subscription balance: The amount of LINK maintained on your Subscription account. Requests from consumer contracts are funded as long as sufficient funds are in the balance, so be sure to maintain sufficient funds in your Subscription balance to pay for the requests and keep your applications running.

- Subscription reservation: The amount of LINK blocked on the Subscription balance. It corresponds to the total LINK amount to be paid by in-flight requests.

- Effective balance: The amount of LINK available on your Subscription account. Effective balance = Subscription balance - Subscription reservation.

- Subscription consumers: Consumer contracts are approved to use funding from your Subscription account while making Chainlink Functions requests. The consumers receive response data in a callback.

Accept ToS

To ensure compliance and governance, Chainlink Functions mandates that any account that manages a subscription must first accept the platform's Terms of Service (ToS). The acceptance is verified by cross-referencing the account with the allowedSenders registry contained within the TermsOfServiceAllowList contract.

<ClickToZoom src="/images/chainlink-functions/subscription/acceptToSEOA.png" />

The acceptance process is initiated via the Chainlink Functions Subscription Manager. After a user accepts the ToS by generating the required signature with their externally owned account (EOA), they transmit proof of acceptance to the TermsOfServiceAllowList contract. Upon successful validation of the proof, the EOA is added to the allowedSenders registry, permitting it to manage subscriptions.

Create subscription

<ClickToZoom
src="/images/chainlink-functions/subscription/createSubscription.png" />

After the ToS is accepted, EOAs can create subscriptions. Upon creation, the FunctionsRouter assigns a unique identifier, Subscription ID.

Note: EOAs can directly interact with the FunctionsRouter contract using their preferred web3 library, such as web3.js or ethers.js.

Fund subscription

<ClickToZoom src="/images/chainlink-functions/subscription/fundSubscription.png" />

You must fund your subscription accounts with enough LINK tokens:

1. Connect your EOA to the Chainlink Functions Subscription Manager.
1. Fund your subscription account. The Chainlink Functions Subscription Manager abstracts the following:
 1. Call transferAndCall on the LINK token contract, transferring LINK tokens along with the Subscription ID in the payload.
 1. The FunctionsRouter contract implements onTokenTransfer: It parses the Subscription ID from the payload and funds the subscription account with the transferred LINK amount.

Note: EOAs can directly interact with the LinkToken contract using their preferred web3 library, such as web3.js or ethers.js.

Add consumer

`<ClickToZoom src="/images/chainlink-functions/subscription/addConsumer.png" />`

You must allowlist your consumers' contracts on your subscription account before they can make Chainlink Functions requests:

1. Connect your EOA to the Chainlink Functions Subscription Manager.
1. Add the address of the consumer contract to the subscription account.
1. The Chainlink Functions Subscription Manager interacts with the FunctionsRouter contract to register the consumer contract address to the subscription account.

Note: EOAs can directly interact with the FunctionsRouter contract using a web3 library, such as web3.js or ethers.js.

Remove consumer

`<ClickToZoom
src="/images/chainlink-functions/subscription/removeConsumer.png" />`

To prevent further Chainlink Functions requests from a given consumer contract, you must remove it from your subscription account:

1. Connect your EOA to the Chainlink Functions Subscription Manager.
1. Remove the address of the consumer contract from the subscription account.
1. The Chainlink Functions Subscription Manager communicates with the FunctionsRouter contract to remove the consumer contract address from the subscription account.

Note: You can still remove consumers from your subscription even if in-flight requests exist. The consumer contract will still receive a callback, and your Subscription Account will be charged.

Note: EOAs can directly interact with the FunctionsRouter contract using a web3 library, such as web3.js or ethers.js.

Cancel subscription

`<ClickToZoom
src="/images/chainlink-functions/subscription/cancelSubscription.png" />`

To cancel a subscription:

1. Connect your EOA to the Chainlink Functions Subscription Manager.
1. Cancel your subscription, providing the Subscription Balance receiver account address. The Chainlink Functions Subscription Manager handles the following processes:
 1. Invokes the cancelSubscription function on the FunctionsRouter contract, deleting the Subscription ID and removing all associated consumers.
 1. Transfers the remaining Subscription Balance to the specified receiver account.

Note: EOAs can directly interact with the FunctionsRouter contract using their preferred web3 library, such as web3.js or ethers.js.

Note: Subscriptions cannot be canceled while there are in-flight requests. Furthermore, any expired requests (requests that have yet to receive a response within 5 minutes) must be timed out before cancellation.

Transferring ownership of a Subscription

Transferring ownership is currently only supported using the Functions Hardhat Starter kit or the Functions Toolkit NPM package:

1. Use the `functions-sub-transfer` command to initiate the transfer of ownership by specifying the new owner's address.
1. As a prerequisite, the prospective owner must be part of the `allowedSenders` registry within the `TermsOfServiceAllowList` contract. This verifies their acceptance of the Chainlink Functions' Terms of Service (ToS).
1. The prospective owner should use the Functions Hardhat Starter kit and run the `functions-sub-accept` command to confirm the ownership transfer.

Note: This guide will be updated as soon as the Chainlink Functions Subscription Manager supports ownership transfers.

```
# billing.mdx:
```

```
---
section: chainlinkFunctions
date: Last Modified
title: "Chainlink Functions Billing"
---

import { Aside } from "@components"
import { TabsContent } from "@components/Tabs"
```

Request costs

To send Chainlink Functions requests, you must maintain a sufficient amount of LINK in your subscription balance. Because Chainlink Functions follows the Request and Receive Data model, billing is done in two steps:

1. During the request step, the cost to fulfill a Chainlink Functions request is estimated and blocked on the subscription balance by adding it to the subscription reservation.
1. During the receive step, the exact fulfillment cost is calculated, then billed to the subscription account, and the subscription reservation is removed.

You can break down total costs into the following components:

- Gas cost: This cost is paid back to the transmitter oracle in LINK for fulfilling the request.
- Premium fees: These fees are paid in LINK to compensate nodes for their work and for the maintenance of the `FunctionsRouter` contract.

Gas cost calculation includes the following variables:

- Gas price: The current gas price fluctuates depending on network conditions.
- Overestimated gas price: refers to a deliberately higher gas price estimation to account for potential price fluctuations between the time a request is made and the time Chainlink Functions fulfills it. By setting the gas price higher than the current one, Chainlink Functions increases the likelihood that the request will be fulfilled, even if gas prices rise unexpectedly in the short term. The overestimation is calculated as a percentage increase over the current gas price.
- Callback gas: The amount of gas used for the callback request. See the Cost Calculation section.
- Callback gas limit: The maximum amount of gas that can be used to call the `handleOracleFulfillment` callback function of the consumer contract in order to provide the response. See the Cost Simulation section.
- Gas overhead: The amount of gas used by the `FunctionsRouter` and `FunctionsCoordinator` contracts. It is an estimate of the total gas cost of fulfilling a request.
- Native to LINK translation: Your subscription balance can be billed only in LINK tokens.
 - Translate the network's native gas tokens to LINK: The total gas cost in native units is translated using the correct Price Feed. For example, on

Sepolia, the translation uses the ETH/LINK Price Feed, and on Polygon, the translation uses POL/LINK Price Feed.

- Fallback Wei to LINK ratio: In the unlikely event that the Native to LINK price data feed is unavailable, the fallback translation is used. You can find this ratio by running the getConfig function in the FunctionsCoordinator contract. See the Supported Networks page to find the contract addresses for each network.

Cost simulation (reservation)

During the request step:

1. The total cost in LINK is estimated using the following formula:

$$\text{total estimated gas cost in LINK} = (\text{Overestimated gas price} \times (\text{Gas overhead} + \text{Callback gas limit})) / \text{Native to LINK translation}$$
$$\text{total estimated cost in LINK} = \text{total estimated gas cost} + \text{premium fees}$$

All networks have USD-denominated premium fees. This means that the fixed premium fee is denominated in USD, but no USD is ever used. The LINK equivalent of this fee is calculated at request time.

1. The total estimated cost is then added to the subscription reservation.

Cost calculation (fulfillment)

When a DON's oracle reports the response, subscription accounts are charged based on the gas amount used in the callback:

1. The total cost in LINK is calculated using the following formula:

$$\text{total gas cost in LINK} = (\text{Gas price} \times (\text{Gas overhead} + \text{Callback gas})) / \text{Native to LINK translation}$$
$$\text{total cost in LINK} = \text{total gas cost} + \text{premium fees}$$

All networks have USD-denominated premium fees. This means that the fixed premium fee is denominated in USD, but no USD is ever used. The LINK equivalent of this fee is calculated at request time.

1. The FunctionsRouter contract performs several accounting movements.

Cost calculation example

This is an example of cost estimation for a request and then cost calculation at fulfillment.

For networks with USD-denominated premium fees, the premium fees are set in USD, but no USD is ever used. The LINK equivalent of the fee is calculated at request time, and then deducted from your subscription in LINK at response time.

Cost Simulation (Reservation)

Parameter	Value
-----	-----
Overestimated gas price	9 gwei
Callback gas limit	300000
Gas overhead	185000
Premium fee	320 cents USD

1. Calculate the total estimated gas cost in LINK, using an overestimated gas price, the gas overhead, and the full callback gas limit:

Gas cost calculation	Total
estimated gas cost	

Overestimated gas price x (Gas overhead + Callback gas limit)	
9 gwei x (300000 + 185000)	4365000
gwei (0.004365 ETH)	

1. Convert the gas cost to LINK using the LINK/ETH feed.

For this example, assume the feed returns a conversion value of 1 0.007 ETH per 1 LINK.

ETH to LINK cost conversion	Total gas cost (LINK)
-----	-----
0.004365 ETH / 0.007 ETH/LINK	0.62 LINK

1. Convert the USD-denominated premium fee to LINK using the LINK/USD feed.

For this example, assume the feed returns a conversion value of \$20.00 USD per 1 LINK:

USD to LINK cost conversion	Premium fee (LINK)
-----	-----
\$3.20 USD / 20.00 USD/LINK	0.16 LINK

1. Add the converted premium fee to get the estimated cost for a subscription reservation:

Adding converted LINK premium	Maximum request cost (LINK)
-----	-----
Total gas cost (LINK) + Premium fee	
0.62 LINK + 0.16 LINK	0.78 LINK

For this example request, 0.78 LINK is reserved from your subscription balance, but not yet deducted. When the request is fulfilled, the exact request cost is deducted. The estimated cost of a request is overestimated to allow for 99% of gas price fluctuation increases in between request and response time.

Cost calculation (fulfillment)

Parameter	Value
-----	-----
Gas price	1.5 gwei
Callback gas	200000
Gas overhead	185000
Premium fee converted to LINK	0.16 LINK

1. Calculate the total gas cost:

Gas cost calculation	Total gas cost
-----	-----
Gas price x (Gas overhead + Callback gas)	
1.5 gwei x (200000 + 185000)	577500 gwei (0.0005775 ETH)

1. Convert the gas cost to LINK using the LINK/ETH feed.

For this example, assume the feed returns a conversion value of 1 0.007 ETH per 1 LINK.

ETH to LINK cost conversion	Total gas cost (LINK)
-----	-----

| 0.0005775 ETH / 0.007 ETH/LINK | 0.0825 LINK |

1. The premium fee was converted from USD to LINK at the time of the request. Add this converted premium fee to get the total cost of a request:

Adding premium fee	Total request cost (LINK)
-----	-----
Total gas cost (LINK) + premium fee	
0.0825 LINK + 0.16 LINK	0.2425 LINK

This example request would cost 0.2425 LINK when it is fulfilled. The subscription reservation for the 0.78 LINK is released, and the actual cost of 0.2425 LINK is deducted from your subscription balance.

Minimum balance for uploading encrypted secrets

If you choose to store the encrypted secrets with the DON (learn more on the Secrets Management page), then one of your subscriptions must maintain a balance greater than the minimum required for uploading encrypted secrets. This balance is blockchain-specific. You can find the specific values for each blockchain on the Supported Networks page.

Note: Uploading encrypted secrets is free of charge. However, to prevent misuse of Chainlink Functions, you are required to maintain a minimum balance in one of the subscriptions owned by your externally-owned account (EOA) before you can upload encrypted secrets to a DON.

Withdrawing funds

To withdraw your LINK balance, you must first cancel your subscription. To prevent misuse of Chainlink Functions, any subscription with fulfilled requests below the request threshold will incur a cancellation fee. Both the request threshold and the cancellation fee are blockchain-specific. You can find their values for each blockchain on the Supported Networks page.

Example 1:

- Request Threshold: Two requests
- Cancellation Fee: 0.5 LINK
- Your Balance: 0.4 LINK
- Number of Fulfilled Requests: One
- Outcome: Canceling your subscription results in a non-refundable balance

Example 2:

- Request Threshold: Two requests
- Cancellation Fee: 0.5 LINK
- Your Balance: 1 LINK
- Number of Fulfilled Requests: One
- Outcome: You will receive 0.5 LINK if you cancel your subscription

Example 3:

- Request Threshold: Two requests
- Cancellation Fee: 0.5 LINK
- Your Balance: 1 LINK
- Number of Fulfilled Requests: Two or more
- Outcome: You will receive 1 LINK if you cancel your subscription. Your subscription will not incur the cancellation fee

index.mdx:

section: chainlinkFunctions

```
date: Last Modified
title: "Chainlink Functions Resources"
isIndex: true
---
```

Topics

- Architecture
- Managing Subscriptions
- Billing
- Supported Networks
- Service Limits

```
# release-notes.mdx:
```

```
---
section: chainlinkFunctions
date: Last Modified
title: "Chainlink Functions Release Notes"
---
```

```
import { Aside } from "@components"
```

Timing out requests manually - 2024-08-21

The Chainlink Functions Subscription Manager now supports timing out requests manually. You can time out requests that are pending for longer than five minutes to unlock your subscription funds.

USD-denominated premium fees on all networks - 2024-07-31

Chainlink Functions now uses USD-denominated fixed premium fees on all supported networks. This means that the premium fees are set in USD, but no USD is ever used. The LINK equivalent of the fee is calculated at request time, and then deducted from your subscription in LINK at response time. See the example cost calculation for more information.

The networks that have just switched from LINK-denominated premium fees to USD-denominated premium fees are:

- Ethereum mainnet and Sepolia testnet
- Arbitrum mainnet
- Avalanche mainnet

Polygon Amoy support - 2024-04-26

Chainlink Functions is available on Polygon Amoy.

Polygon testnet support - 2024-04-13

The Mumbai network has stopped producing blocks, so example code will not function on this network. Check again soon for updates about future testnet support on Polygon.

Base Mainnet support - 2024-04-09

Chainlink Functions is available on Base Mainnet.

USD-denominated premium fees and new testnets - 2024-03-22

Chainlink Functions is available as an open beta on the following testnets:

- BASE Sepolia

- Optimism Sepolia

Both of these networks have USD-denominated fixed premium fees. This means that the premium fees are set in USD, but no USD is ever used. The LINK equivalent of the fee is calculated at request time, and then deducted from your subscription in LINK at response time. See the example cost calculation for more information.

Module imports supported on mainnet - 2024-01-12

- You can use external module imports with Chainlink Functions source code on mainnet networks. See the Using Imports with Functions tutorial to see an example of how to import and use imported modules with your Functions source code. This feature requires the Functions Toolkit NPM package v0.2.7 or later.

Arbitrum Mainnet support - 2024-01-10

Chainlink Functions is available on Arbitrum Mainnet.

Module imports and new testnets - 2023-12-15

- You can use external module imports with Chainlink Functions source code on testnet networks. See the Using Imports with Functions tutorial to see an example of how to import and use imported modules with your Functions source code. This feature requires the Functions Toolkit NPM package v0.2.7 or later.

This feature is available only on testnets. Modules will not import or execute on Functions requests for mainnet networks at this time.

- Chainlink Functions is available on the Arbitrum Sepolia testnet.

Open Beta - 2023-09-29

- Chainlink Functions is available as an open beta on the following blockchains:

- Ethereum :

- Ethereum Mainnet
- Ethereum Sepolia

- Polygon:

- Polygon Mainnet
- Polygon Mumbai

- Avalanche:

- Avalanche Mainnet
- Avalanche Fuji

See the supported networks page for more information.

- New features:

- You must accept the Chainlink Functions Terms of Service (ToS) before using Chainlink Functions. The ToS must be accepted by subscriptions owners. Once accepted, the ToS is transitive to all contracts belong the subscription, so your end-users don't have to accept the ToS to interact with your contracts. Read this guide to learn more.

- The Chainlink Functions Subscription Manager is available at functions.chain.link. The Functions Subscription Manager lets you manage your subscriptions.

- Chainlink Functions uses threshold encryption to handle users' encrypted secrets. Read the secrets conceptual page to learn more.

- Users can host their encrypted secrets within the DON. This hosting method is called DON-hosted. Read the secrets conceptual page to learn more.

- JavaScript source code can only use vanilla Deno. Read the JavaScript code

API reference to learn more.

- Chainlink Functions contracts are part of the @chainlink/contracts npm package. Read the FunctionsClient and FunctionsRequest API references.
- Use the Functions npm package in your own JavaScript or TypeScript project to make requests to the Chainlink Functions Decentralized Oracle Network (DON). Try the getting-started guide to learn more.
- Make sure to check the service limits page as the limits have been adapted. Additionally, you can contact us to increase the limits for your Chainlink Function.

Functions playground - 2023-07-14

Use the Functions Playground to simulate Chainlink Functions within your browser.

Closed beta - New testnet - 2023-05-05

New testnet added:

- Avalanche Fuji

See the supported networks page for more information.

Closed beta - 2023-03-01

Chainlink Functions is available on the following testnets:

- Ethereum Sepolia
- Polygon Mumbai

secrets.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "Secrets Management"
---
```

```
import { Aside } from "@components"
```

```
<Aside type="caution">
```

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

```
</Aside>
```

Chainlink Functions enables users to fetch data from HTTP(s) APIs and perform custom computation. To enable access to APIs that require authentication credentials, Chainlink Functions allows users to provide encrypted secret values, which can be used when the DON executes a Functions request. These values can only be decrypted via a multi-party decryption process called threshold decryption. This means that every node can only decrypt the secrets with participation from other DON nodes.

These encrypted secret values can either be uploaded to the DON or hosted at a remote URL. Then, a reference to the encrypted secrets can be used when making a Functions request.

Threshold encryption

Using a single private key to decrypt user secrets poses a considerable security risk. If any node with this key becomes malicious or compromised, it can decrypt all accessible user secrets.

To address this vulnerability, we have integrated a feature known as threshold encryption. The characteristics of this enhanced security measure include:

1. Decentralized Nodes Deployment: The system operates on a Decentralized Oracle Network (DON) with N nodes.

1. Master Secret Key Distribution: Instead of centralizing the private key, the master secret key (MSK) is partitioned into shares. Each node within the DON possesses a distinct share of this MSK.

1. The system enforces two thresholds:

- Liveness: This feature ensures the system's availability and fault tolerance. It can tolerate up to F Byzantine nodes (nodes that can act arbitrarily or maliciously) while remaining operational. The system adheres to the mathematical relationship $3F + 1 \leq N$. This means the total number of nodes N must be at least three times the number of faulty or malicious nodes F, plus one more.

- Decryption security: Considering that the MSK is distributed across multiple nodes, safeguarding it is essential. To recover the master secret key (MSK), an attacker must corrupt a number of K nodes, adhering to the relationship:

$$F < K \leq 2F + 1.$$

1. Encryption Process: A public key is associated with the MSK. Users use this public key to encrypt their secrets, generating ciphertext. Note: The encrypted secrets are never stored onchain.

1. Threshold Decryption Mechanism: Decrypting any given ciphertext mandates the collaboration of at least K out of N key shares, where:

$$F < K \leq 2F + 1$$

For decryption to initiate, a minimum of K nodes must concurrently process a user request bearing the identical ciphertext.

Hosting methods

There are two methods for sharing encrypted secrets with the DON:

- DON-hosted (Location.DONHosted): Users send the encrypted secrets (ciphertext) to the DON through the secrets endpoint, as depicted in the request and receive diagram. Note: To use this capability, one of the subscriptions owned by your externally-owned account (EOA) must maintain a minimum balance. Read the minimum balance for uploading encrypted secrets section to learn more.

- User-hosted (Location.Remote): Some users prefer to manage the storage and lifecycle of their secrets, such as deleting the encrypted secrets once their request is fulfilled. In this setup, users host the encrypted secrets (ciphertext) on a publicly accessible HTTP(s) endpoint and then encrypt the URL with the DON public key before sharing the encrypted URL with the DON. Note: After the request has been addressed, the user is responsible for removing their stored ciphertext.

service-limits.mdx:

section: chainlinkFunctions

date: Last Modified

title: "Chainlink Functions Service Limits"

import { Aside } from "@components"

<Aside type="note" title="Increase the limits">

 Contact us to
increase the limits for your Chainlink
Function.

</Aside>

Item	Description
Limits	

Supported languages	Supported language of the Source code that is submitted in the Chainlink Functions request
JavaScript compatible with Deno runtime and module imports	
Maximum requests in flight	The maximum number of simultaneous Chainlink Functions requests that are not fulfilled yet by the DON
Limited by the effective balance	
Maximum callback gas limit	The maximum amount of gas that you can set for Chainlink Functions to fulfill your callback function
300,000	
Maximum subscriptions	Maximum subscriptions that you can create
unbounded	
Maximum consumer contracts per subscription	The maximum number of consumer contracts that you can add to a subscription
100	
Maximum duration of a request (Request fulfillment timeout)	The maximum duration of a request is the time allowed for processing a Chainlink Functions request from start to finish, covering all steps such as reading from the chain, executing code, and posting back on chain. If processing takes longer, the request times out, and no charges are applied to your subscription for that request. In such cases, the initially locked estimated request costs must be manually requested to be returned to your subscription balance 5 minutes
Maximum request size	The maximum size of a Chainlink Request. This includes the source code, arguments, and secrets
30 kilobytes	
Maximum returned value size	The maximum size of the value that your Function can return
256 bytes	
Maximum source code execution time	The maximum amount of time that a source code can run
10 seconds	

Maximum memory allocated to the source code amount of memory allocated to your source code during execution 128 megabytes	The maximum
HTTP - Maximum queries number of HTTP requests that your source code can make 5	The maximum
HTTP - query timeout duration of an HTTP request before it times out 9 seconds	The maximum
HTTP - Maximum URL length the HTTP URL 2048 characters	Length of
HTTP - Maximum request length size of an HTTP request, including the request body and HTTP headers 30 kilobytes	The maximum
HTTP - Maximum response length size of an HTTP response 2 megabytes	The maximum
Max size of request payload CBOR size of an encoded request 30 kilobytes	The maximum
Don-hosted secrets - Maximum Time to live (TTL) duration for which a secret hosted on a DON remains valid before it gets deleted Mainnets: 3 months (2160 hours)Testnets: 3 days (72 hours)	The maximum
External module imports - Maximum size size in MB that each imported module can be 10 MB	The maximum
External module imports - Number of imports number of external module imports allowed in each request 100 imports	The maximum

simulation.mdx:

```

---
section: chainlinkFunctions
date: Last Modified
title: "Simulate your Functions"
---
```

```
import { ClickToZoom, Aside } from "@components"
```

Before making a Chainlink Functions request from your smart contract, it is always a good practice to simulate the source code off-chain to make any adjustments or corrections.

Currently, there are several options for simulating a request:

- Chainlink Functions playground.

- Chainlink Functions Hardhat Starter Kit: Use the `npx hardhat functions-simulate-script` command to simulate your Functions JavaScript source code.

- Chainlink Functions Toolkit NPM package: Import this NPM package into your JavaScript/TypeScript project, then use the `simulateScript` function to simulate your Functions JavaScript source code.

Chainlink Functions playground

<Aside type="caution">

The Chainlink Functions playground can simulate Chainlink Functions within the browser. There may be differences to the real Chainlink Functions environment. For instance, you are limited to vanilla JavaScript and cannot use all the features of Chainlink Functions, such as imports of external modules. To access all the features, use the Chainlink Functions Hardhat Starter Kit or Chainlink Functions Toolkit NPM package.

</Aside>

To use the Chainlink Functions Playground, enter any source code, arguments, and secrets you would like to use. Click the Run code button to view the output.

<ClickToZoom src="/images/chainlink-functions/functions-playground.png" />

Chainlink Functions Hardhat Starter Kit

This repository comes preconfigured with Hardhat and the Chainlink Functions Toolkit NPM package, allowing you to quickly get started with Functions.

To simulate:

1. In a terminal, clone the `functions-hardhat-starter-kit` repository and change to the `functions-hardhat-starter-kit` directory.

```
shell
git clone https://github.com/smartcontractkit/functions-hardhat-starter-kit
&& \
cd functions-hardhat-starter-kit
```

1. Run `npm install` to install the dependencies.

```
shell
npm install
```

1. For simulation, you don't need to set up the environment variables. Run `npx hardhat functions-simulate-script` to simulate the `calculation-example.js` JavaScript source code. Note: This example calculates the continuously compounding interest for a given principal amount over one month. It takes the principal amount and annual percentage yield (APY) as input arguments and uses Euler's number to compute the total amount after interest.

```
shell
npx hardhat functions-simulate-script
```

Result:

```
text
secp256k1 unavailable, reverting to browser version
```

Response returned by script during local simulation: 1003757

```
<Aside type="note">
  The JavaScript source code is set up in the
  Functions-request-config.js
  file. To simulate your own JavaScript source code, modify source to
  replace ./calculation-example.js with the
  relative path to your file.
</Aside>
```

Chainlink Functions Toolkit NPM package

Follow the Simple Computation guide to learn how to import the Chainlink Functions Toolkit NPM package into your JavaScript project to simulate and execute a Functions request. You can read the Examine the code section for a detailed description of the code example.

```
# subscriptions.mdx:

---
section: chainlinkFunctions
date: Last Modified
title: "Managing CL Functions Subscriptions"
---

import { Aside, ClickToZoom } from "@components"
```

```
<Aside type="note" title="Mainnet Beta">
Chainlink Functions is available on mainnet only as a BETA preview to ensure
that this new platform is robust and secure for developers. While in BETA,
developers must follow best practices and not use the BETA for any mission-
critical application or secure any value. Chainlink Functions is likely to
evolve and improve. Breaking changes might occur while the service is in BETA.
Monitor these docs to stay updated on feature improvements along with interface
and contract changes.
```

```
</Aside>
```

```
<Aside type="note">
Chainlink Functions is a self-service solution. You are responsible for
independently reviewing any code and API dependencies that you submit in a
request. Community-created code examples might not be audited, so you must
independently review this code before you use it.
```

Chainlink Functions is offered "as is" and "as available" without conditions or warranties of any kind. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs from Functions due to issues in your code or downstream issues with API dependencies. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. Users are responsible for complying with the licensing agreements for all data providers that they connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account or subscription.

```
</Aside>
```

The Chainlink Functions Subscription Manager is available [here](#). The Functions Subscription Manager lets you create a subscription, add consumers to it, remove consumers from it, fund it with LINK, and delete it.

When you connect to the Subscription Manager, choose the correct network, then click connect wallet.

Subscriptions

You use a Chainlink Functions subscription to pay for, manage, and track Functions requests.

Create a subscription

1. Open `functions.chain.link` and click Create Subscription:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/frontend-landing-
wallet-connected.jpg"
  alt="Chainlink Functions subscription landing page"
/>
```

1. Provide an email address and an optional subscription name:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/create-
subscription.jpg"
  alt="Chainlink Functions create subscription"
/>
```

1. The first time that you interact with the Subscription Manager using your EOA, you have to approve the Terms of Service (ToS):

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/acceptTos.jpg"
  alt="Chainlink Functions accept ToS"
/>
```

1. When you approve the ToS, a MetaMask popup appears that asks you to approve the subscription creation:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/approve-create-
subscription.jpg"
  alt="Chainlink Functions approve subscriptions creation"
/>
```

1. After the subscription is created, you are asked to sign a message in MetaMask to link the subscription name and email address to your subscription:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/sign-message-tos.jpg"
  alt="Chainlink Functions sign message ToS"
/>
```

```
{" "}
```

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/sign-message-tos-
metamask.jpg"
  alt="Chainlink Functions sign message ToS MetaMask"
/>
```

1. After the subscription is created, fund it with LINK:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
created-add-funds.jpg"
  alt="Chainlink Functions subscription add funds"
/>
```

1. After funding, you can add a consumer to it:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
created-add-consumer.jpg"
  alt="Chainlink Functions subscription add consumer"
/>
```

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
created-add-consumer-2.jpg"
  alt="Chainlink Functions subscription add consumer"
/>
```

1. After creation, you can fetch the details of your subscription:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/subscription-
details-after-creation.jpg"
  alt="Chainlink Functions subscription details"
/>
```

Fund a subscription

1. Open your subscription details and click Actions then click Fund subscription:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/your-subscription-
click-fund.jpg"
  alt="Chainlink Functions click fund"
/>
```

1. Fund your subscription. For instance, 0.1 LINK:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/your-subscription-
add-fund.jpg"
  alt="Chainlink Functions add fund to your subscription"
/>
```

1. A MetaMask popup appears, and you are asked to confirm the transaction. After you confirm the transaction, a confirmation screen appears:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/your-subscription-
funded.jpg"
  alt="Chainlink Functions subscription funded"
/>
```

Add a consumer contract to a subscription

1. Open your subscription details and click Add Consumer:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/your-subscription-
add-consumer.jpg"
  alt="Chainlink Functions add consumer"
/>
```

1. Fill in the consumer address:

```
<ClickToZoom
```

```
src="/images/chainlink-functions/tutorials/subscription/your-subscription-
fill-consumer.jpg"
alt="Chainlink Functions provider consumer address"
/>
```

1. A MetaMask popup appears, and you are asked to confirm the transaction. After you confirm the transaction, a confirmation screen appears:

```
<ClickToZoom
src="/images/chainlink-functions/tutorials/subscription/your-subscription-
consumer-added.jpg"
alt="Chainlink Functions consumer added"
/>
```

Remove a consumer contract from a subscription

1. Open your subscription details and click the consumer you want to remove, then Remove Consumer:

```
<ClickToZoom
src="/images/chainlink-functions/tutorials/subscription/your-subscription-
remove-consumer.jpg"
alt="Chainlink Functions remove consumer"
/>
```

1. A MetaMask popup appears, and you are asked to confirm the transaction.

```
<ClickToZoom
src="/images/chainlink-functions/tutorials/subscription/your-subscription-
remove-consumer-confirm.jpg"
alt="Chainlink Functions confirm the removal of the consumer"
/>
```

```
<ClickToZoom
src="/images/chainlink-functions/tutorials/subscription/your-subscription-
remove-consumer-confirm2.jpg"
alt="Chainlink Functions confirm the removal of the consumer in MetaMask"
/>
```

1. After you confirm the transaction, a confirmation screen appears:

```
<ClickToZoom
src="/images/chainlink-functions/tutorials/subscription/your-subscription-
consumer-removed.jpg"
alt="Chainlink Functions consumer removed"
/>
```

Cancel a subscription

Note: You cannot cancel a subscription if there are in-flight requests. In-flight requests are requests that still need to be fulfilled.

1. Open your subscription details, click Actions, then Cancel subscription:

```
<ClickToZoom
src="/images/chainlink-functions/tutorials/subscription/your-subscription-
cancel.jpg"
alt="Chainlink Functions cancel subscription"
/>
```

1. Fill in the receiver address of the remaining funds:

```
<ClickToZoom
src="/images/chainlink-functions/tutorials/subscription/your-subscription-
```

```
cancel2.jpg"
  alt="Chainlink Functions cancel subscription"
/>
```

1. A MetaMask popup appears, and you are asked to confirm the transaction.

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/your-subscription-
cancel-confirm.jpg"
  alt="Chainlink Functions confirm the removal of the subscription"
/>
```

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/your-subscription-
cancel-confirm2.jpg"
  alt="Chainlink Functions confirm the removal of the subscription in
MetaMask"
/>
```

1. After you confirm the transaction, a confirmation screen appears:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/subscription/your-subscription-
cancelled.jpg"
  alt="Chainlink Functions subscription canceled"
/>
```

Time out pending requests manually

The Subscription Manager provides the option to time out requests manually. You can time out requests that have been pending for longer than five minutes to unlock your subscription funds.

1. Open your subscription details and scroll to the Pending section. If a request has been pending longer than five minutes, its status displays as Time out required. Click the link within the red banner that says Time out request:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/request-timeout/1-request-
eligible-for-timeout.png"
  alt="Pending Functions request eligible for timeout"
/>
```

If you have multiple requests that are eligible for timeout, the Time out request link in the red banner times out all the requests at once.

Alternatively, you can time out an individual request. Open the Actions menu for that request, and click Time out request:

```
{" "}
```

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/request-timeout/individual-
request-timeout.png"
  alt="Individual request timeout"
/>
```

1. A MetaMask popup appears, and you are asked to confirm the transaction:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/request-timeout/2-approve-
timeout-transaction.png"
  alt="Prompt to approve timeout transaction"
/>
```


1. After you confirm the transaction, a confirmation screen appears:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/request-timeout/3-timeout-
successful-confirmation.png"
  alt="Timeout transaction confirmation"
/>
```

1. The subscription details page displays another confirmation in the upper right corner showing that the request has been timed out successfully:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/request-timeout/4-subscription-
page-updates.png"
  alt="Request timed out dialog"
/>
```

1. The timed out request appears in the Failed fulfillments tab of the History section:

```
<ClickToZoom
  src="/images/chainlink-functions/tutorials/request-timeout/5-failed-
fulfillments.png"
  alt="Chainlink Functions subscription canceled"
/>
```

abi-decoding.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "Return multiple responses and decode them in your smart contract"
metadata:
  linkToWallet: true
whatsnext:
  {
    "Try out the Chainlink Functions Tutorials":
"/chainlink-functions/tutorials",
    "Read the Architecture to understand how Chainlink Functions operates":
"/chainlink-functions/resources/architecture",
  }
---
```

```
import { Aside, CopyText, CodeSample } from "@components"
import { Tabs } from "@components/Tabs"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

In the Using Imports with Functions tutorial, we explored the fundamentals of module imports. This tutorial will teach you how to use the Ethers library encode function to perform ABI encoding of several responses. Then, you will use the ABI specifications in Solidity to decode the responses in your smart contract.

<Aside type="caution">

Users are fully responsible for any dependencies their JavaScript source code imports. Chainlink is not responsible

for any imported dependencies and provides no guarantees of the validity, availability or security of any libraries a

user chooses to import or the repositories from which these dependencies are downloaded. Developers are advised to

fully vet any imported dependencies or avoid dependencies altogether to avoid

any risks associated with a compromised library or a compromised repository from which the dependency is downloaded.

Prerequisites

This tutorial assumes you have completed the Using Imports with Functions tutorial. Also, check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager. If your subscription runs out of LINK, follow the Fund a Subscription guide.

In this tutorial, you will use a different Chainlink Functions consumer contract, which shows how to use ABI decoding to decode the response received from Chainlink Functions:

1. Open the FunctionsConsumerDecoder.sol contract in Remix.

```
<CodeSample src="samples/ChainlinkFunctions/FunctionsConsumerDecoder.sol" showButtonOnly={true} />
```

1. Compile the contract.

1. Open MetaMask and select the Ethereum Sepolia network.

1. In Remix under the Deploy & Run Transactions tab, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Ethereum Sepolia.

1. Under the Deploy section, fill in the router address for your specific blockchain. You can find this address on the Supported Networks page. For Ethereum Sepolia, the router address is `<CopyText text="0xb83E47C2bC239B3bf370bc41e1459A34b41238D0" code/>`.

1. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Ethereum Sepolia.

1. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy the contract address.

1. Add your consumer contract address to your subscription on Ethereum Sepolia.

Tutorial

This tutorial demonstrates using the ethers library to interact with smart contract functions through a JSON RPC provider. It involves calling the latestRoundData, decimals, and description functions of a price feed contract based on the AggregatorV3Interface.

After retrieving the necessary data, the guide shows how to use ABI encoding to encode these responses into a single hexadecimal string and then convert this string to a Uint8Array. This step ensures compliance with the Chainlink Functions API requirements, which specify that the source code must return a Uint8Array representing the bytes for on-chain use.

You can locate the scripts used in this tutorial in the examples/12-abi-encoding directory.

To run the example:

1. Make sure you have correctly set up your environment first. If you haven't already, follow the Set up your environment section of the Using Imports with Functions tutorial.

1. Open the file request.js, located in the 12-abi-encoding folder.

1. Replace the consumer contract address and the subscription ID with your own values.

```
javascript
const consumerAddress = "0x5fC6e53646CC53f0C3575fd2c71b5056c4823f5c" //
```

[illegible]

}

 $\hat{\alpha}_e.$

```
0x0000000000000000000000000000000000000000000063c3570cc840000000000000000  
00000000000000000000000000000000000000000000661969f700000000000000000000000000  
000000000000000000000000000000000000000000008000000000000000000000000000000000  
000000000000000000000080000000000000000000000000000000000000000000000000000000  
09425443202f2055534400000000000000000000000000000000000000000000
```

```

    @... Fetched BTC / USD price: 6855664389252 (updatedAt: 1712941559) (decimals:
8) (description: BTC / USD)

```

The output of the example gives you the following information:

```
- Your request is first run on a sandbox environment to ensure it is
correctly configured.
- The fulfillment costs are estimated before making the request.
- Your request was successfully sent to Chainlink Functions. The transaction
in this example is
0x5618089ec9b5e662ec72c81241d78cb6daa135ecc3fa3a33032d910e3b47c2b1, and the
request ID is
0xdf22fa28c81a3ea78f356334b6d28d969e953009fae8ece4fe544f2eb466419b.
```

- The DON successfully fulfilled your request. The total cost was: 0.282344694329387405 LINK.

```
- The consumer contract received a response in hexadecimal string with a
value of
```

[illegible]

```
- The script calls the consumer contract to fetch the decoded values and then
logs them to the console. The output is Fetched BTC / USD price: 6855664389252
(updatedAt: 1712941559) (decimals: 8) (description: BTC / USD).
```

Examine the code

FunctionsConsumerDecoder.sol

```
<CodeSample src="samples/ChainlinkFunctions/FunctionsConsumerDecoder.sol" />
```

This Solidity contract is similar to the `FunctionsConsumer.sol` contract used in the `Using Imports with Functions` tutorial. The main difference is the processing of the response in the `fulfillRequest` function:

```
- It uses Solidity abi.decode to decode the response to retrieve the answer,
updatedAt, decimals, and description.
```

```

solidity
(
    uint256 answer,
    uint256 updatedAt,
    uint8 decimals,
    string memory description
) = abi.decode(response, (uint256, uint256, uint8, string));

```

- Then stores the decoded values in the contract state.

```

solidity
sanswer = answer;
supdatedAt = updatedAt;
sdecimals = decimals;
sdescription = description;

```

JavaScript example

source.js

The Decentralized Oracle Network will run the JavaScript code. The code is self-explanatory and has comments to help you understand all the steps.

```
<ChainlinkFunctions section="deno-importe-notes" />
```

The example source.js file is similar to the one used in the Using Imports with Functions tutorial. It uses a JSON RPC call to the latestRoundData, decimals, and description functions of a Chainlink Data Feed. It then uses the ethers library to encode the response of these functions into a single hexadecimal string.

```

javascript
const encoded = ethers.AbiCoder.defaultAbiCoder().encode(
  ["uint256", "uint256", "uint8", "string"],
  [dataFeedResponse.answer, dataFeedResponse.updatedAt, decimals, description]
)

```

Finally, it uses the ethers library getBytes to convert the hexadecimal string to a Uint8Array:

```

javascript
return ethers.getBytes(encoded)

```

request.js

This explanation focuses on the request.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs: Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsDecoder.json: The abi of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerDecoder contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```

javascript
const consumerAddress = "0x5fC6e53646CC53f0C3575fd2c71b5056c4823f5c" // REPLACE
this with your Functions consumer address
const subscriptionId = 139 // REPLACE this with your subscription ID

```

The primary function that the script executes is `makeRequestSepolia`. This function consists of five main parts:

1. Definition of necessary identifiers:

- `routerAddress`: Chainlink Functions router address on Sepolia.
- `donId`: Identifier of the DON that will fulfill your requests on Sepolia.
- `explorerUrl`: Block explorer URL of the Sepolia testnet.
- `source`: The source code must be a string object. That's why we use `fs.readFileSync` to read `source.js` and then call `toString()` to get the content as a string object.
- `args`: During the execution of your function, These arguments are passed to the source code.
- `gasLimit`: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

1. Simulating your request in a local sandbox environment:

- Use `simulateScript` from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.bytes` in this example).

1. Estimating the costs:

- Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost`.
- The response is returned in Juels (1 LINK = 10¹⁸ Juels). Use the `ethers.utils.formatEther` utility function to convert the output to LINK.

1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `sendRequest` function of your consumer contract.

1. Waiting for the response:

- Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.bytes` in this example).

1. Read the decoded response:

- Call the `answer`, `supdatedAt`, `sdecimals`, and `sdescription` functions of your consumer contract to fetch the decoded values.
- Log the decoded values to the console.

Handling complex data types with ABI Encoding and Decoding

This section details the process of encoding complex data types into `Uint8Array` typed arrays to fulfill the Ethereum Virtual Machine (EVM) data handling requirements for transactions and smart contract interactions. It will then outline the steps for decoding these byte arrays to align with corresponding structures defined in Solidity.

Consider a scenario where a contract needs to interact with a data structure

that encapsulates multiple properties, including nested objects:

```
json
{
  "id": 1,
  "metadata": {
    "description": "Decentralized Oracle Network",
    "awesome": true
  }
}
```

Transferring and storing this kind of structured data requires encoding it into a format (array of 8-bit unsigned integers) that smart contracts can accept and process.

Encoding in JavaScript

Because Chainlink Functions supports important external modules, you can import a web3 library such as ethers.js and perform encoding. To encode complex data structures, you can use the `defaultAbiCoder.encode` function from the ethers.js library. The function takes two arguments:

- An array of Solidity data types.
- The corresponding data in JavaScript format.

and returns the encoded data as a hexadecimal string.

Here's how you can encode the aforementioned complex data:

```
javascript
const { ethers } = await import("npm:ethers@6.10.0") // Import ethers.js v6.10.0

const abiCoder = ethers.AbiCoder.defaultAbiCoder()

// Define the data structure
const complexData = {
  id: 1,
  metadata: {
    description: "Decentralized Oracle Network",
    awesome: true,
  },
}

// Define the Solidity types for encoding
const types = ["tuple(uint256 id, tuple(string description, bool awesome) metadata)"]

// Encoding the data
const encodedData = abiCoder.encode(types, [complexData])
```

After encoding the data, it's necessary to format it as a `Uint8Array` array for smart contract interactions and blockchain transactions. In Solidity, the data type for byte arrays data is `bytes`. However, when working in a JavaScript environment, such as when using the ethers.js library, the equivalent data structure is a `Uint8Array`.

The ethers.js library provides the `getBytes` function to convert encoded hexadecimal strings into a `Uint8Array`:

```
javascript
return ethers.getBytes(encodedData) // Return the encoded data converted into a Uint8Array
```

Decoding in Solidity

The encoded data can be decoded using the `abi.decode` function. To decode the data, you'll need to handle the decoding in your `fulfillRequest` function:

```
solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

contract DataDecoder {
    // Example of a structure to hold the complex data
    struct Metadata {
        string description;
        bool awesome;
    }

    struct ComplexData {
        uint256 id;
        Metadata metadata;
    }

    // ... other contract functions (including the send request function)

    // Fulfill function (callback function)
    function fulfillRequest(bytes32 requestId, bytes memory response, bytes memory
err) internal override {
        // Decode the response
        ComplexData memory metadata = abi.decode(response, (ComplexData));
        // ... rest of the function
    }
}
```

api-multiple-calls.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "Call Multiple Data Sources"
---
```

```
import { Aside } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how make multiple API calls from your smart contract to a Decentralized Oracle Network. After OCR completes offchain computation and aggregation, the DON returns the asset price to your smart contract. This example returns the BTC/USD price.

This guide assumes that you know how to build HTTP requests and how to use secrets. Read the API query parameters and API use secrets guides before you follow the example in this document.

To build a decentralized asset price, send a request to the DON to fetch the price from many different API providers. Then, calculate the median price. The API providers in this example are:

- CoinMarket
- CoinGecko
- CoinPaprika

<Aside type="caution">

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

</Aside>

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

</Aside>

<ChainlinkFunctions section="prerequisites-guides" />

Tutorial

This tutorial is configured to get the median BTC/USD price from multiple data sources. For a detailed explanation of the code example, read the [Examine the code](#) section.

You can locate the scripts used in this tutorial in the `examples/8-multiple-apis` directory.

1. Make sure your subscription has enough LINK to pay for your requests. Also, you must maintain a minimum balance to upload encrypted secrets to the DON (Read the [minimum balance for uploading encrypted secrets](#) section to learn more). You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager. If your subscription runs out of LINK, follow the [Fund a Subscription](#) guide. This guide recommends maintaining at least 2 LINK within your subscription.

1. Get a free API key from CoinMarketCap and note your API key.

1. Run `npx env-enc set` to add an encrypted `COINMARKETCAPAPIKEY` to your `.env.enc` file.

```
shell
npx env-enc set
```

To run the example:

1. Open the file `request.js`, which is located in the `8-multiple-apis` folder.
1. Replace the consumer contract address and the subscription ID with your own values.

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

1. Make a request:

```
shell
```

```
node examples/8-multiple-apis/request.js
```

The script runs your function in a sandbox environment before making an onchain transaction:

```

text
$ node examples/8-multiple-apis/request.js
secp256k1 unavailable, reverting to browser version
Start simulation...
Simulation result {
  capturedTerminalOutput: 'Median Bitcoin price: 66822.81\n',
  responseBytesHexString:
'0x000000000000000000000000000000000000000000000000000000000000000065f6a9'
}
â€¦ Decoded response to uint256: 6682281n

Estimate request costs...
Fulfillment cost estimated to 1.104471544715335 LINK

Make request...
Upload encrypted secret to gateways https://01.functions-
gateway.testnet.chain.link/,https://02.functions-gateway.testnet.chain.link/.
slotId 0. Expiration in minutes: 15

â€¦ Secrets uploaded properly to gateways https://01.functions-
gateway.testnet.chain.link/,https://02.functions-gateway.testnet.chain.link/!
Gateways response: { version: 1712949659, success: true }

â€¦ Functions request sent! Transaction hash
0x8defda7d48f91efa4f7bfa8e7d99f115a4e1d71882852ee6e91f438542d840ec. Waiting for
a response...
See your request in the explorer
https://sepolia.etherscan.io/tx/0x8defda7d48f91efa4f7bfa8e7d99f115a4e1d71882852e
e6e91f438542d840ec

â€¦ Request
0xff18de309a7845ef99b042d008aa3c5e67c51e649b771cbaab7dd96fada66e27 successfully
fulfilled. Cost is 0.25590956997723491 LINK.Complete reponse: {
  requestId:
'0xff18de309a7845ef99b042d008aa3c5e67c51e649b771cbaab7dd96fada66e27',
  subscriptionId: 2303,
  totalCostInJuels: 255909569977234910n,
  responseBytesHexString:
'0x000000000000000000000000000000000000000000000000000000000000000065f6a9',
  errorString: '',
  returnDataBytesHexString: '0x',
  fulfillmentCode: 0
}

â€¦ Decoded response to uint256: 6682281n

```

The output of the example gives you the following information:

- Your request is first run on a sandbox environment to ensure it is correctly configured.
- The fulfillment costs are estimated before making the request.
- The encrypted secrets were uploaded to the secrets endpoint `https://01.functions-gateway.testnet.chain.link/user`.
- Your request was successfully sent to Chainlink Functions.
- The DON successfully fulfilled your request. The total cost was: `0.25590956997723491 LINK`.
- The consumer contract received a response in bytes with a value of

args.

- Make the HTTP calls.
- Read the asset price from each response.
- Calculate the median of all the prices.
- Return the result as a buffer using the Functions.encodeUint256 helper function. Because solidity doesn't support decimals, multiply the result by 100 and round the result to the nearest integer. Note: Read this article if you are new to Javascript Buffers and want to understand why they are important.

request.js

This explanation focuses on the request.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsClient.json: The ABI of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerExample contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is makeRequestSepolia. This function can be broken into six main parts:

1. Definition of necessary identifiers:

- routerAddress: Chainlink Functions router address on Sepolia.
- donId: Identifier of the DON that will fulfill your requests on Sepolia.
- gatewayUrls: The secrets endpoint URL to which you will upload the encrypted secrets.
- explorerUrl: Block explorer URL of the Sepolia testnet.
- source: The source code must be a string object. That's why we use fs.readFileSync to read source.js and then call toString() to get the content as a string object.
- args: During the execution of your function, These arguments are passed to the source code. The args value is ["1", "bitcoin", "btc-bitcoin"]. These arguments are BTC IDs at CoinMarketCap, CoinGecko, and Coinpaprika. You can adapt args to fetch other asset prices.
- secrets: The secrets object that will be encrypted.
- slotIdNumber: Slot ID at the DON where to upload the encrypted secrets.
- expirationTimeMinutes: Expiration time in minutes of the encrypted secrets.
- gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

1. Simulating your request in a local sandbox environment:

- Use `simulateScript` from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

1. Estimating the costs:

- Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost` function.
- The response is returned in Juels (1 LINK = 10¹⁸ Juels). Use the `ethers.utils.formatEther` utility function to convert the output to LINK.

1. Encrypt the secrets, then upload the encrypted secrets to the DON. This is done in two steps:

- Initialize a `SecretsManager` instance from the Functions NPM package, then call the `encryptSecrets` function.
- Call the `uploadEncryptedSecretsToDON` function of the `SecretsManager` instance. This function returns an object containing a success boolean as long as version, the secret version on the DON storage. Note: When making the request, you must pass the slot ID and version to tell the DON where to fetch the encrypted secrets.

1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `sendRequest` function of your consumer contract.

1. Waiting for the response:

- Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

api-post-data.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "POST Data to an API"
---
```

```
import { Aside } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how to send a request to a Decentralized Oracle Network to call the Countries information GraphQL API. After OCR completes offchain computation and aggregation, it returns the name, capital, and currency for the specified country to your smart contract. Because the endpoint is a GraphQL API, write a function that sends a GraphQL query in a POST HTTP method.

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data

provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

<Aside type="note" title="Maximum response size">

You can return any number of responses as long as they are encoded in a bytes response. The maximum response size that you can return is 256 bytes.

</Aside>

<Aside type="caution" title="Side effects">

Building non-idempotent requests, such as sending an email or storing data on the cloud, is currently not recommended.

An HTTP method is idempotent if the intended effect on the server when you make a single request is the same as the

effect when you make several identical requests. Each oracle node runs the same computation in the Offchain Reporting

protocol. If your Chainlink Function makes non-idempotent requests, it

will cause redundant requests such as sending multiple emails or storing the same data multiple times.

</Aside>

<ChainlinkFunctions section="prerequisites-guides" />

Tutorial

This tutorial is configured to get the country name, capital, and currency from countries.trevorblades.com in one request. For a detailed explanation of the code example, read the Examine the code section.

You can locate the scripts used in this tutorial in the examples/4-post-data directory.

To run the example:

1. Open the file request.js, which is located in the 4-post-data folder.
1. Replace the consumer contract address and the subscription ID with your own values.

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

1. Make a request:

```
shell
node examples/4-post-data/request.js
```

The script runs your function in a sandbox environment before making an onchain transaction:

```
text
$ node examples/4-post-data/request.js
secp256k1 unavailable, reverting to browser version
Start simulation...
Simulation result {
  capturedTerminalOutput: 'Get name, capital and currency for country code:
JP\n' +
  'HTTP POST Request to https://countries.trevorblades.com/\n' +
  'country response { country: { name: "Japan", capital: "Tokyo", currency:
```

```

"JPY" } } \n',
  responseBytesHexString:
'0x7b226e616d65223a224a6170616e222c226361706974616c223a22546f6b796f222c226375727
2656e6379223a224a5059227d'
}
â€¦ Decoded response to string:
{"name":"Japan","capital":"Tokyo","currency":"JPY"}

Estimate request costs...
Fulfillment cost estimated to 1.007671833192655 LINK

Make request...

â€¦ Functions request sent! Transaction hash
0x5a315eeebea90f4828d176906d13dd3133e8a8d9afa912b1f8c34e90e775d081. Waiting for
a response...
See your request in the explorer
https://sepolia.etherscan.io/tx/0x5a315eeebea90f4828d176906d13dd3133e8a8d9afa912
b1f8c34e90e775d081

â€¦ Request
0xc760ee5ca5c73999ca9c4ce426b9d2d44eab4429d3110276e57c445537ad5ddd successfully
fulfilled. Cost is 0.257726519296170771 LINK. Complete response: {
  requestId:
'0xc760ee5ca5c73999ca9c4ce426b9d2d44eab4429d3110276e57c445537ad5ddd',
  subscriptionId: 2303,
  totalCostInJuels: 257726519296170771n,
  responseBytesHexString:
'0x7b226e616d65223a224a6170616e222c226361706974616c223a22546f6b796f222c226375727
2656e6379223a224a5059227d',
  errorString: '',
  returnDataBytesHexString: '0x',
  fulfillmentCode: 0
}

â€¦ Decoded response to string:
{"name":"Japan","capital":"Tokyo","currency":"JPY"}

```

The output of the example gives you the following information:

- Your request is first run on a sandbox environment to ensure it is correctly configured.
- The fulfillment costs are estimated before making the request.
- Your request was successfully sent to Chainlink Functions. The transaction in this example is 0x5a315eeebea90f4828d176906d13dd3133e8a8d9afa912b1f8c34e90e775d081 and the request ID is 0xc760ee5ca5c73999ca9c4ce426b9d2d44eab4429d3110276e57c445537ad5ddd.
- The DON successfully fulfilled your request. The total cost was: 0.257726519296170771 LINK.
- The consumer contract received a response in bytes with a value of 0x7b226e616d65223a224a6170616e222c226361706974616c223a22546f6b796f222c2263757272656e6379223a224a5059227d. Decoding it offchain to string gives you a result:

```
{"name":"Japan","capital":"Tokyo","currency":"JPY"}.
```

Examine the code

FunctionsConsumerExample.sol

```
<ChainlinkFunctions section="functions-consumer" />
```

JavaScript example

source.js

The Decentralized Oracle Network will run the JavaScript code. The code is self-explanatory and has comments to help you understand all the steps.

```
<ChainlinkFunctions section="deno-importe-notes" />
```

This JavaScript source code uses `Functions.makeHttpRequest` to make HTTP requests. To request the JP country information, the source code calls the `https://countries.trevorblades.com/` URL and provides the query data in the HTTP request body. If you read the `Functions.makeHttpRequest` documentation, you see that you must provide the following parameters:

- url: `https://countries.trevorblades.com/`
- data (HTTP body):

```
{
  query: {\
    country(code: "${countryCode}") { \
      name \
      capital \
      currency \
    } \
  },
}
```

To check the expected API response:

- In your browser, open the countries GraphQL playground:
`https://countries.trevorblades.com/`
- Write this query:

```
{
  country(code: "JP") {
    name
    capital
    currency
  }
}
```

- Click on play to get the answer :

```
json
{
  "data": {
    "country": {
      "name": "Japan",
      "capital": "Tokyo",
      "currency": "JPY"
    }
  }
}
```

The main steps of the scripts are:

- Fetch the `countryCode` from `args`.
- Construct the HTTP object `countryRequest` using `Functions.makeHttpRequest`.

- Run the HTTP request.
- Read the country name, capital, and currency from the response.
- Construct a JSON object:

```
javascript
const result = {
  name: countryData.country.name,
  capital: countryData.country.capital,
  currency: countryData.country.currency,
}
```

- Convert the JSON object to a JSON string using `JSON.stringify(result)`. This step is mandatory before encoding string to bytes.
 - Return the result as a buffer using the `Functions.string` helper function.
- Note: Read this article if you are new to Javascript Buffers and want to understand why they are important.

`request.js`

This explanation focuses on the `request.js` script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- `path` and `fs` : Used to read the source file.
- `ethers`: Ethers.js library, enables the script to interact with the blockchain.
- `@chainlink/functions-toolkit`: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- `@chainlink/env-enc`: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- `../abi/functionsClient.json`: The abi of the contract your script will interact with. Note: The script was tested with this `FunctionsConsumerExample` contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBED20A71397064D9" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is `makeRequestSepolia`. This function consists of five main parts:

1. Definition of necessary identifiers:

- `routerAddress`: Chainlink Functions router address on Sepolia.
- `donId`: Identifier of the DON that will fulfill your requests on Sepolia.
- `explorerUrl`: Block explorer URL of the Sepolia testnet.
- `source`: The source code must be a string object. That's why we use `fs.readFileSync` to read `source.js` and then call `toString()` to get the content as a string object.
- `args`: During the execution of your function, These arguments are passed to the source code. The `args` value is `["JP"]`, which fetches country data for Japan.
- `gasLimit`: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

1. Simulating your request in a local sandbox environment:

- Use `simulateScript` from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.string` in this example).

1. Estimating the costs:

- Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost`.
- The response is returned in Juels (1 LINK = 10¹⁸ Juels). Use the `ethers.utils.formatEther` utility function to convert the output to LINK.

1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `sendRequest` function of your consumer contract.

1. Waiting for the response:

- Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.string` in this example).

api-query-parameters.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "Call an API with HTTP Query Parameters"
---
```

```
import { Aside } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how to send a request to a Decentralized Oracle Network to call the Cryptocompare GET /data/pricemultifull API. After OCR completes offchain computation and aggregation, it returns the asset price for ETH/USD to your smart contract. This guide also shows you how to configure HTTP query parameters to request different asset prices.

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

</Aside>

<ChainlinkFunctions section="prerequisites-guides" />

Tutorial

This tutorial is configured to get the ETH/USD price. For a detailed explanation of the code example, read the [Examine the code](#) section.

To run the example:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

```
}
```

â€¦ Decoded response to uint256: 342162n

The output of the example gives you the following information:

- Your request is first run on a sandbox environment to ensure it is correctly configured.
- The fulfillment costs are estimated before making the request.
- Your request was successfully sent to Chainlink Functions. The transaction in this example is 0xbbe473ccc6593b6f3baf30fd66b2329b05a32fe0321a319d09142f4b9ba4547c and the request ID is 0xe55201188012e3ec198427937f7897729999ab7b287207ff8f0c157a9662e5f0.
- The DON successfully fulfilled your request. The total cost was: 0.249819373001796045 LINK.
- The consumer contract received a response in bytes with a value of 0x0053892. Decoding it offchain to uint256 gives you a result: 342162.

Examine the code

FunctionsConsumerExample.sol

```
<ChainlinkFunctions section="functions-consumer" />
```

JavaScript example

source.js

The Decentralized Oracle Network will run the JavaScript code. The code is self-explanatory and has comments to help you understand all the steps.

```
<ChainlinkFunctions section="deno-importe-notes" />
```

This JavaScript source code uses Functions.makeHttpRequest to make HTTP requests. To request the ETH/USD price, the source code calls the <https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD> URL. If you read the Functions.makeHttpRequest documentation, you see that you must provide the following parameters:

- url: <https://min-api.cryptocompare.com/data/pricemultifull>
- params: The query parameters object:

```
{
  fsyms: fromSymbol,
  tsyms: toSymbol
}
```

To check the expected API response, you can directly paste the following URL in your browser <https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD> or run the curl command in your terminal:

```
bash
curl -X 'GET' \
  'https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD' \
  -H 'accept: application/json'
```

The response should be similar to the following example:

```

{/ prettier-ignore /}
json
{
  "RAW": {
    "ETH": {
      "USD": {
        "TYPE": "5",
        "MARKET": "CCCAGG",
        "FROMSYMBOL": "ETH",
        "TOSYMBOL": "USD",
        "FLAGS": "2049",
        "PRICE": 2867.04,
        "LASTUPDATE": 1650896942,
        "MEDIAN": 2866.2,
        "LASTVOLUME": 0.16533939,
        "LASTVOLUMETO": 474.375243849,
        "LASTTRADEID": "1072154517",
        "VOLUMEDAY": 195241.78281014622,
        "VOLUMEDAYTO": 556240560.4621655,
        "VOLUME24HOUR": 236248.94641103,
        ...
      }
    }
  }
}

```

The price is located at RAW,ETH,USD,PRICE.

The main steps of the scripts are:

- Fetch fromSymbol and toSymbol from args.
- Construct the HTTP object cryptoCompareRequest using Functions.makeHttpRequest.
- Make the HTTP call.
- Read the asset price from the response.
- Return the result as a buffer using the Functions.encodeUint256 helper function. Because solidity doesn't support decimals, multiply the result by 100 and round the result to the nearest integer. Note: Read this article if you are new to Javascript Buffers and want to understand why they are important.

request.js

This explanation focuses on the request.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsClient.json: The abi of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerExample contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```

javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID

```

The primary function that the script executes is `makeRequestSepolia`. This function consists of five main parts:

1. Definition of necessary identifiers:

- `routerAddress`: Chainlink Functions router address on Sepolia.
- `donId`: Identifier of the DON that will fulfill your requests on Sepolia.
- `explorerUrl`: Block explorer URL of the Sepolia testnet.
- `source`: The source code must be a string object. That's why we use `fs.readFileSync` to read `source.js` and then call `toString()` to get the content as a string object.
- `args`: During the execution of your function, These arguments are passed to the source code. The `args` value is `["ETH", "USD"]`, which fetches the current ETH/USD price. You can adapt `args` to fetch another asset price. See the [CryptoCompare API docs](#) to get the list of supported symbols.
- `gasLimit`: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

1. Simulating your request in a local sandbox environment:

- Use `simulateScript` from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

1. Estimating the costs:

- Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost`.
- The response is returned in Juels (1 LINK = 10¹⁸ Juels). Use the `ethers.utils.formatEther` utility function to convert the output to LINK.

1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `sendRequest` function of your consumer contract.

1. Waiting for the response:

- Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

api-use-secrets-gist.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "Using User-hosted (gist) Secrets in Requests"
---
```

```
import { Aside } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how to send a request to a Decentralized Oracle Network to call the Coinmarketcap API. After OCR completes offchain computation and aggregation, it returns the BTC/USD asset price to your smart contract. Because the API requires you to provide an API key, this guide will also show you how to encrypt, sign your API key, and share the encrypted secret offchain with a Decentralized Oracle Network (DON).

The encrypted secrets are never stored onchain. This tutorial uses the threshold encryption feature. The encrypted secrets are stored by the user as gists. Read the Secrets Management page to learn more.

Read the Using Secrets in Requests tutorial before you follow the steps in this example. This tutorial uses the same example but with a slightly different process:

1. Instead of uploading the encrypted secrets to the DON, you will host your encrypted secrets as gist.
1. Encrypt the gist URL.
1. Include the encrypted gist URL in your Chainlink Functions request.

<Aside type="caution">

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

</Aside>

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

</Aside>

<ChainlinkFunctions section="prerequisites-guides" />

Tutorial

This tutorial is configured to get the BTC/USD price with a request that requires API keys. For a detailed explanation of the code example, read the Examine the code section.

You can locate the scripts used in this tutorial in the examples/6-use-secrets-gist directory.

1. Get a free API key from CoinMarketCap and note your API key.

1. The request.js example stores encrypted secrets as gists to share them offchain with the Decentralized Oracle Network. To allow the request.js script to write gists on your behalf, create a github fine-grained personal access token.

1. Visit Github tokens settings page.
1. Click on Generate new token.
1. Provide a name to your token and define the expiration date.
1. Under Account permissions, enable Read and write for Gists. Note: Do not

1. Click on Generate token and copy your fine-grained personal access token.
1. Run `npx env-enc set` to add an encrypted GITHUBAPITOKEN and COINMARKETCAPAPIKEY to your `.env.enc` file.

To run the example:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

```

âœ… Request
0x37f860fba46af84b84f5ce48efbb7c6ebbfbb2ecde5063621f695bbd1c2547975 successfully
fulfilled. Cost is 0.237283011969506455 LINK.Complete reponse: {
  requestId:
'0x37f860fba46af84b84f5ce48efbb7c6ebbfbb2ecde5063621f695bbd1c2547975',

```


- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsClient.json: The abi of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerExample contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is makeRequestSepolia. This function can be broken into six main parts:

1. Definition of necessary identifiers:

- routerAddress: Chainlink Functions router address on the Sepolia testnet.
- donId: Identifier of the DON that will fulfill your requests on the Sepolia testnet.
- explorerUrl: Block explorer url of the Sepolia testnet.
- source: The source code must be a string object. That's why we use fs.readFileSync to read source.js and then call toString() to get the content as a string object.
- args: During the execution of your function, These arguments are passed to the source code. The args value is ["1", "USD"], which fetches the BTC/USD price.
- secrets: The secrets object that will be encrypted.
- gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

1. Simulating your request in a local sandbox environment:

- Use simulateScript from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package decodeResult function and ReturnType enum to decode the response to the expected returned type (ReturnType.uint256 in this example).

1. Estimating the costs:

- Initialize a SubscriptionManager from the Functions NPM package, then call the estimateFunctionsRequestCost function.
- The response is returned in Juels (1 LINK = 10¹⁸ Juels). Use the ethers.utils.formatEther utility function to convert the output to LINK.

1. Encrypt the secrets, then create a gist containing the encrypted secrets object. This is done in two steps:

- Initialize a SecretsManager instance from the Functions NPM package, then call the encryptSecrets function.
- Call the createGist utility function from the Functions NPM package to create a gist.
- Call the encryptedSecretsUrls function of the SecretsManager instance. This function encrypts the gist URL. Note: The encrypted URL will be sent to the DON when making a request.

1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `sendRequest` function of your consumer contract.

1. Waiting for the response:

- Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).
- Call the `deleteGist` utility function from the Functions NPM package to delete the gist.

api-use-secrets-offchain.mdx:

```
---
section: chainlinkFunctions
date: Last Modified
title: "Using User-hosted Secrets in Requests"
---
```

```
import { Aside } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how to send a request to a Decentralized Oracle Network to call the Coinmarketcap API. After OCR completes offchain computation and aggregation, it returns the BTC/USD asset price to your smart contract. Because the API requires you to provide an API key, this guide will also show you how to encrypt, sign your API key, and share the encrypted secret offchain with a Decentralized Oracle Network (DON).

The encrypted secrets are never stored onchain. This tutorial uses the threshold decryption feature. This tutorial shows you how to share encrypted secrets offchain with a Decentralized Oracle Network (DON) using a storage platform such as AWS S3, Google Drive, IPFS, or any other service where the DON can fetch secrets via HTTP. Read the [Secrets Management](#) page to learn more.

Read the [Using User-hosted \(gist\) Secrets in Requests](#) tutorial before you follow the steps in this example. This tutorial uses the same example but with a slightly different process:

1. Instead of relying on storing the encrypted secrets on gist, you will host your encrypted secrets on AWS S3.
1. Include the encrypted secrets in an `offchain-secrets.json` file.
1. Host the secrets file offchain (AWS S3).
1. Encrypt the S3 HTTPS URL .
1. Include the encrypted URL in your Chainlink Functions request.

<Aside type="caution">

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

</Aside>

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

</Aside>

<ChainlinkFunctions section="prerequisites-guides" />

Tutorial

This tutorial is configured to get the BTC/USD price with a request that requires API keys. For a detailed explanation of the code example, read the [Examine the code](#) section.

You can locate the scripts used in this tutorial in the `examples/7-use-secrets-url` directory.

1. Get a free API key from [CoinMarketCap](#).

1. Run `npx env-enc set` to add an encrypted `COINMARKETCAPAPIKEY` to your `.env.enc` file.

```
shell
npx env-enc set
```

1. Prepare the store for your encrypted secrets file.

1. Create a AWS free tier account.

1. Follow these steps to create a AWS S3 bucket. Choose a name for your bucket, set ACLs enabled, and turn off Block all public access.

Build Offchain Secrets

Before you make a request, prepare the secrets file and host it offchain:

1. Encrypt the secrets and store them in the `offchain-secrets.json` file using the `gen-offchain-secrets` script of the `7-use-secrets-url` folder.

```
bash
node examples/7-use-secrets-url/gen-offchain-secrets.js
```

Example:

```
text
$ node examples/7-use-secrets-url/gen-offchain-secrets.js
secp256k1 unavailable, reverting to browser version
Encrypted secrets object written to /functions-examples/offchain-secrets.json
```

1. Follow these steps to upload the file `offchain-secrets.json` to your AWS S3 bucket.

1. To make the file publically accessible without authentication:

1. Find the file in the bucket list, and click on it to open the object overview.

1. Click on the Permissions tab to display the Access control list (ACL).

1. Click on Edit.

1. Set Everyone (public access) Objects read, then confirm. This action makes

1. Note the object URL.
1. To verify that the URL is publicly readable without authentication, open a new browser tab and copy/paste the object URL in the browser location bar. After you hit Enter , the browser will display the content of your encrypted secrets file.

Send a Request

1. Open the file `request.js`, which is located in the `7-use-secrets-url` folder.
1. Replace the consumer contract address and the subscription ID with your own values.

1. Replace the secretsUrls with your AWS S3 URL:

1. Make a request:

The script runs your function in a sandbox environment before making an onchain transaction:

```
text
$ node examples/7-use-secrets-url/request.js
secp256k1 unavailable, reverting to browser version
Encrypted secrets object written to /Users/crystalgomes/smart-contract-
examples/functions-examples/offchain-secrets.json
crystalgomes@MB-CY16VK6DPG functions-examples % node examples/7-use-secrets-
url/request.js
```

[illegible]

Estimate request costs...
Fulfillment cost estimated to 1.018348822253235 LINK

Make request...

Encrypt the URLs..

â€¦ Functions request sent! Transaction hash
0xadc0db0ddea7b9836b86a9c9e008bc97d47e5f92b0dcec9694d3944d0065c789. Waiting for
a response...
See your request in the explorer
<https://sepolia.etherscan.io/tx/0xadc0db0ddea7b9836b86a9c9e008bc97d47e5f92b0dcec9694d3944d0065c789>

[illegible]

```
âœ… Decoded response to uint256: 6820970n
```

The output of the example gives you the following information:

- ```
- Your request is first run on a sandbox environment to ensure it is
correctly configured.
- The fulfillment costs are estimated before making the request.
- The AWS S3 URL is encrypted before sending it in the request.
- Your request was successfully sent to Chainlink Functions. The transaction
in this example is
0xadcd0db0ddea7b9836b86a9c9e008bc97d47e5f92b0dcec9694d3944d0065c789 and the
request ID is
0xb308ca293859dab47d8848578291e687a0d9373274d1451a9c9667dc4bba5fca.
```

- [illegible]

Examine the code

FunctionsConsumerExample.sol

```
<ChainlinkFunctions section="functions-consumer" />
```

### JavaScript example

source.js

The JavaScript code is similar to the [Using Secrets in Requests](#) tutorial.

```
gen-offchain-secrets.js
```

This explanation focuses on the `gen-offchain-secrets.js` script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to encrypts your secrets. After encryption, the script saves the encrypted secrets on a local file, `offchain-secrets.json`. You can then upload the file to your storage of choice (AWS S3 in this example).

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.

The primary function that the script executes is generateOffchainSecretsFile. This function can be broken into three main parts:

#### 1. Definition of necessary identifiers:

- routerAddress: Chainlink Functions router address on Sepolia.
- donId: Identifier of the DON that will fulfill your requests on Sepolia.
- secrets: The secrets object.
- Initialization of ethers signer and provider objects. The Chainlink NPM package uses the signer to sign the encrypted secrets with your private key.

#### 1. Encrypt the secrets:

- Initialize a SecretsManager instance from the Chainlink Functions NPM package.
- Call the encryptSecrets function from the created instance to encrypt the secrets.

#### 1. Use the fs library to store the encrypted secrets on a local file, offchain-secrets.json.

request.js

This explanation focuses on the request.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsClient.json: The abi of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerExample contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is makeRequestSepolia. This function can be broken into six main parts:

#### 1. Definition of necessary identifiers:

- routerAddress: Chainlink Functions router address on Sepolia.
- donId: Identifier of the DON that will fulfill your requests on Sepolia.

- explorerUrl: Block explorer URL of the Sepolia testnet.
- source: The source code must be a string object. That's why we use `fs.readFileSync` to read `source.js` and then call `toString()` to get the content as a string object.
- args: During the execution of your function, These arguments are passed to the source code. The args value is `["1", "USD"]`, which fetches the BTC/USD price.
- secrets: The secrets object. Note: Because we are sharing the URL of the encrypted secrets with the DON, the secrets object is only used during simulation.
- secretsUrls: The URL of the encrypted secrets object.
- gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

#### 1. Simulating your request in a local sandbox environment:

- Use `simulateScript` from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

#### 1. Estimating the costs:

- Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost` function.
- The response is returned in Juels (1 LINK = 10<sup>18</sup> Juels). Use the `ethers.utils.formatEther` utility function to convert the output to LINK.

#### 1. Encrypt the secrets, then create a gist containing the encrypted secrets object. This is done in two steps:

- Initialize a `SecretsManager` instance from the Functions NPM package, then call the `encryptSecrets` function.
- Call the `encryptedSecretsUrls` function of the `SecretsManager` instance. This function encrypts the secrets URL. Note: The encrypted URL will be sent to the DON when making a request.

#### 1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `sendRequest` function of your consumer contract.

#### 1. Waiting for the response:

- Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

# api-use-secrets.mdx:

```

section: chainlinkFunctions
date: Last Modified
title: "Using DON-hosted Secrets in Requests"

```



```
import { Aside } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how to send a request to a Decentralized Oracle Network to call the Coinmarketcap API. After OCR completes offchain computation and aggregation, it returns the BTC/USD asset price to your smart contract. Because the API requires you to provide an API key, this guide will also show you how to encrypt, sign your API key, and share the encrypted secret with a Decentralized Oracle Network (DON).

The encrypted secrets are never stored onchain. This tutorial uses the threshold encryption feature. The encrypted secrets are stored with the DON. Read the Secrets Management page to learn more.

<Aside type="caution">

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

</Aside>

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

</Aside>

<ChainlinkFunctions section="prerequisites-guides" />

## Tutorial

This tutorial is configured to get the BTC/USD price with a request that requires API keys. For a detailed explanation of the code example, read the Examine the code section.

You can locate the scripts used in this tutorial in the examples/5-use-secrets-threshold directory.

1. Get a free API key from CoinMarketCap and note your API key.

1. Run `npx env-enc set` to add an encrypted `COINMARKETCAPAPIKEY` to your `.env.enc` file.

```
shell
npx env-enc set
```

1. Make sure your subscription has enough LINK to pay for your requests. Also, you must maintain a minimum balance to upload encrypted secrets to the DON (Read the minimum balance for uploading encrypted secrets section to learn more). You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager. If your subscription runs out of LINK, follow the Fund a Subscription guide. This guide recommends maintaining at least 2 LINK within your subscription.

To run the example:

1. Open the file `request.js`, which is located in the `5-use-secrets-threshold` folder.
1. Replace the consumer contract address and the subscription ID with your own values.

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

- ## 1. Make a request:

```
shell
node examples/5-use-secrets-threshold/request.js
```

The script runs your function in a sandbox environment before making an onchain transaction:

[illegible]

```
âœ… Decoded response to uint256: 6675505n
```

The output of the example gives you the following information:

- ```
- Your request is first run on a sandbox environment to ensure it is
correctly configured.
- The fulfillment costs are estimated before making the request.
- The encrypted secrets were uploaded to the secrets endpoint
https://01.functions-gateway.testnet.chain.link/user.
- Your request was successfully sent to Chainlink Functions. The transaction
in this example is
0xcac39aeea98651f307da185aed387314c453272d185e58b26b3bb399b82a90b6 and the
request ID is
0xbc09de04f4dd39fa78d4b00b7ab4d2f4a37d8b9a8edf97df5c86061175b9d9c3.

- The DON successfully fulfilled your request. The total cost was:
0.23730142355580769 LINK.
- The consumer contract received a response in bytes with a value of
0x000000000000000000000000000000000000000000000000000000000000000065dc31. Decoding it
offchain to uint256 gives you a result: 6675505. The median BTC price is
66755.05 USD.
```

Examine the code

FunctionsConsumerExample.sol

```
<ChainlinkFunctions section="functions-consumer" />
```

JavaScript example

source.js

The Decentralized Oracle Network will run the JavaScript code. The code is self-explanatory and has comments to help you understand all the steps.

```
<ChainlinkFunctions section="deno-importe-notes" />
```

This JavaScript source code uses `Functions.makeHttpRequest` to make HTTP requests. To request the BTC asset price, the source code calls the `https://pro-api.coinmarketcap.com/v1/cryptocurrency/quotes/latest/` URL. If you read the `Functions.makeHttpRequest` documentation, you see that you must provide the following parameters:

- ```
- url: https://pro-api.coinmarketcap.com/v1/cryptocurrency/quotes/latest
- headers: This is an HTTP headers object set to "X-CMCPROAPIKEY":
secrets.apiKey. The apiKey is passed in the secrets, see request.
- params: The query parameters object:
```

```
{
 convert: currencyCode,
 id: coinMarketCapCoinId
}
```

To check the expected API response, run the curl command in your terminal:

```
bash
curl -X 'GET' \
 'https://pro-api.coinmarketcap.com/v1/cryptocurrency/quotes/latest?
id=1&convert=USD' \
```

```
-H 'accept: application/json' \
-H 'X-CMCPROAPIKEY: REPLACEWITHYOURAPIKEY'
```

The response should be similar to the following example:

```
{/ prettier-ignore /}
json
{
 '...',
 "data": {
 "1": {
 "id": 1,
 "name": "Bitcoin",
 "symbol": "BTC",
 "slug": "bitcoin",
 '...',
 "quote": {
 "USD": {
 "price": 23036.068560170934,
 "volume24h": 33185308895.694683,
 "volumechange24h": 24.8581,
 "percentchange1h": 0.07027098,
 "percentchange24h": 1.79073805,
 "percentchange7d": 10.29859656,
 "percentchange30d": 38.10735851,
 "percentchange60d": 39.26624921,
 "percentchange90d": 11.59835416,
 "marketcap": 443982488416.99316,
 "marketcapdominance": 42.385,
 "fullydilutedmarketcap": 483757439763.59,
 "tvl": null,
 "lastupdated": "2023-01-26T18:27:00.000Z"
 }
 }
 }
 }
}
```

The price is located at data,1,quote,USD,price.

The main steps of the scripts are:

- Fetch the currencyCode and coinMarketCapCoinId from args.
- Construct the HTTP object coinMarketCapRequest using Functions.makeHttpRequest.
- Make the HTTP call.
- Read the asset price from the response.
- Return the result as a buffer using the helper function: Functions.encodeUint256. Note: Because solidity doesn't support decimals, we multiply the result by 100 and round the result to the nearest integer. Note: Read this article if you are new to Javascript Buffers and want to understand why they are important.

request.js

This explanation focuses on the request.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsClient.json: The abi of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerExample contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is makeRequestSepolia. This function can be broken into six main parts:

#### 1. Definition of necessary identifiers:

- routerAddress: Chainlink Functions router address on Sepolia.
- donId: Identifier of the DON that will fulfill your requests on Sepolia.
- gatewayUrls: The secrets endpoint URL to which you will upload the encrypted secrets.
- explorerUrl: Block explorer URL of the Sepolia testnet.
- source: The source code must be a string object. That's why we use fs.readFileSync to read source.js and then call toString() to get the content as a string object.
- args: During the execution of your function, These arguments are passed to the source code. The args value is ["1", "USD"], which fetches the BTC/USD price.
- secrets: The secrets object that will be encrypted.
- slotIdNumber: Slot ID at the DON where to upload the encrypted secrets.
- expirationTimeMinutes: Expiration time in minutes of the encrypted secrets.
- gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

#### 1. Simulating your request in a local sandbox environment:

- Use simulateScript from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package decodeResult function and ReturnType enum to decode the response to the expected returned type (ReturnType.uint256 in this example).

#### 1. Estimating the costs:

- Initialize a SubscriptionManager from the Functions NPM package, then call the estimateFunctionsRequestCost function.
- The response is returned in Juels (1 LINK = 10<sup>18</sup> Juels). Use the ethers.utils.formatEther utility function to convert the output to LINK.

#### 1. Encrypt the secrets, then upload the encrypted secrets to the DON. This is done in two steps:

- Initialize a SecretsManager instance from the Functions NPM package, then call the encryptSecrets function.
- Call the uploadEncryptedSecretsToDON function of the SecretsManager instance. This function returns an object containing a success boolean as long

as version, the secret version on the DON storage. Note: When making the request, you must pass the slot ID and version to tell the DON where to fetch the encrypted secrets.

#### 1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the sendRequest function of your consumer contract.

#### 1. Waiting for the response:

- Initialize a ResponseListener from the Functions NPM package and then call the listenForResponseFromTransaction function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package decodeResult function and ReturnType enum to decode the response to the expected returned type (ReturnType.uint256 in this example).

```
automate-functions-custom-logic.mdx:
```

```

section: chainlinkFunctions
date: Last Modified
title: "Automate your Functions (Custom Logic Automation)"

```

```
import { Aside, ClickToZoom, CopyText, CodeSample } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how to use Chainlink Automation to automate your Chainlink Functions. Automation is essential when you want to trigger the same function regularly, such as fetching weather data daily or fetching an asset price on every block.

Read the Automate your Functions (Time-based Automation) tutorial before you follow the steps in this example. This tutorial explains how to trigger your functions using an Automation compatible contract.

After you deploy and set up your contract, Chainlink Automation triggers your function on every block.

<Aside type="caution">

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

</Aside>

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

</Aside>

<ChainlinkFunctions section="prerequisites-guides-with-automation" />

## Tutorial

This tutorial is configured to get the median BTC/USD price from multiple data sources on every block. Read the [Examine the code](#) section for a detailed explanation of the code example.

You can locate the scripts used in this tutorial in the `examples/10-automate-functions` directory.

1. Make sure to understand the [API multiple calls](#) guide.

1. Make sure your subscription has enough LINK to pay for your requests. Also, you must maintain a minimum balance to upload encrypted secrets to the DON (Read the [minimum balance for uploading encrypted secrets](#) section to learn more). You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager. If your subscription runs out of LINK, follow the [Fund a Subscription](#) guide. This guide recommends maintaining at least 2 LINK within your subscription.

1. Get a free API key from [CoinMarketCap](#) and note your API key.

1. Run `npx env-enc set` to add an encrypted `COINMARKETCAPAPIKEY` to your `.env.enc` file.

```
shell
npx env-enc set
```

## Deploy a Custom Automated Functions Consumer contract

<Aside type="caution">

When using Chainlink Automation, developers should be mindful of the risks of Automation attempting to perform upkeeps

indefinitely if a previous upkeep fails to update due to reversion. To learn more, read the [Automation best practices](#).

</Aside>

The consumer contract for Custom Automated Functions is different from the consumer in other tutorials. Deploy the `CustomAutomatedFunctionsConsumerExample` contract on Ethereum Sepolia:

1. Open the `CustomAutomatedFunctionsConsumerExample.sol` in Remix.

```
<CodeSample
src="samples/ChainlinkFunctions/CustomAutomatedFunctionsConsumerExample.sol"
showButtonOnly={true} />
```

1. Compile the contract.

1. Open MetaMask and select the Ethereum Sepolia network.

1. In Remix under the `Deploy & Run Transactions` tab, select `Injected Provider - MetaMask` in the Environment list. Remix will use the MetaMask wallet to communicate with Ethereum Sepolia.

1. Under the `Deploy` section, fill in the router address for your specific blockchain. You can find this address on the [Supported Networks](#) page. For Ethereum Sepolia, the router address is `<CopyText text="0xb83E47C2bC239B3bf370bc41e1459A34b41238D0" code/>`.

1. Click the `Deploy` button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Ethereum Sepolia.

1. After you confirm the transaction, the contract address appears in the

Deployed Contracts list. Copy your contract address.

Add your Consumer contract to your Functions subscription

Add your contract as an approved consumer contract to your Functions subscription using the Chainlink Functions Subscription Manager.

Configure your Consumer contract

Configure the request details by calling the `updateRequest` function. This step stores the encoded request (source code, reference to encrypted secrets if any, arguments), gas limit, subscription ID, and job ID in the contract storage (see [Examine the code](#)). To do so, follow these steps:

1. On a terminal, change directories to the `10-automate-functions` directory.
1. Open `updateRequest.js` and replace the consumer contract address and the subscription ID with your own values:

```
javascript
const consumerAddress = "0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

1. Run the `updateRequest.js` script to update your Functions consumer contract's request details.

Configure Chainlink Automation

The consumer contract that you deployed is designed to be used with a custom logic automation. Follow the instructions in the [Register a Custom Logic Upkeep](#) guide to register your deployed contract using the Chainlink Automation App. Use the following upkeep settings:

- Trigger: Custom logic
- Target contract address: The address of the Chainlink Functions consumer contract that you deployed
- Name: Give your upkeep a name
- Check data: Leave this field blank
- Gas limit: `<CopyText text="1000000" code/>`
- Starting balance (LINK): `<CopyText text="1" code/>`

You can leave the other settings at their default values for the example in this tutorial.

At this stage, your Functions consumer contract is configured to get the median Bitcoin price on every block.

`<Aside type="note" title="Monitor your balances">`  
There are two balances that you must monitor:

- Your subscription balance: Your balance will be charged each time your Chainlink Functions is fulfilled. If your balance is insufficient, your contract cannot send requests. Automating your Chainlink Functions means they will be regularly triggered, so monitor and fund your subscription account regularly. You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager.
- Your upkeep balance: You can check this balance on the Chainlink Automation App. The upkeep balance pays Chainlink Automation Network to send your requests according to your provided time interval. Chainlink Automation will not trigger your requests if your upkeep balance runs low.

`</Aside>`



## Check Result

Go to the Chainlink Automation App and connect to the Sepolia testnet. Your upkeep will be listed under My upkeeps:

<ClickToZoom src="/images/chainlink-functions/tutorials/automation/cl-automation-custom-home.jpg" />

Click on your upkeep to fetch de details:

<ClickToZoom src="/images/chainlink-functions/tutorials/automation/myupkeep-details-custom.jpg" />

On your terminal, run the readLatest to read the latest received response:

1. Open readLatest.js and replace the consumer contract address with your own values:

```
javascript
const consumerAddress = "0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA" //
REPLACE this with your Functions consumer address
```

1. Run the readLatest script.

```
shell
node examples/10-automate-functions/readLatest.js
```

Output Example:

```
text
$ node examples/10-automate-functions/readLatest.js
secp256k1 unavailable, reverting to browser version
Last request ID is
0xf4ae51b028ded52d3376810cd97f02e2b1dff424bb78d45730820fefc0b8060
â€¦ Decoded response to uint256: 6688012n
```

## Clean up

After you finish the guide, cancel your upkeep in the Chainlink Automation App and withdraw the remaining funds. After you cancel the upkeep, there is a 50-block delay before you can withdraw the funds.

Examine the code

CustomAutomatedFunctionsConsumer.sol

<ChainlinkFunctions section="custom-automated-functions-consumer" />

source.js

The JavaScript code is similar to the one used in the Call Multiple Data Sources tutorial.

updateRequest.js

The JavaScript code is similar to the one used in the Automate your Functions (Time-based Automation) tutorial.

readLatest.js

The JavaScript code is similar to the one used in the Automate your Functions

(Time-based Automation) tutorial.

```
automate-functions.mdx:
```

```

section: chainlinkFunctions
date: Last Modified
title: "Automate your Functions (Time-based Automation)"

```

```
import { Aside, ClickToZoom, CopyText, CodeSample } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how to use Chainlink Automation to automate your Chainlink Functions. Automation is essential when you want to trigger the same function regularly, such as fetching weather data daily or fetching an asset price on every block.

Read the API multiple calls tutorial before you follow the steps in this example. This tutorial uses the same example but with an important difference:

- You will deploy AutomatedFunctionsConsumerExample.sol instead of the FunctionsConsumerExample contract.

After you deploy and set up your contract, Chainlink Automation triggers your function according to a time schedule.

<Aside type="caution">

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

</Aside>

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

</Aside>

<ChainlinkFunctions section="prerequisites-guides-with-automation" />

## Tutorial

This tutorial is configured to get the median BTC/USD price from multiple data sources according to a time schedule. For a detailed explanation of the code example, read the Examine the code section.

You can locate the scripts used in this tutorial in the examples/10-automate-functions directory.

1. Make sure to understand the API multiple calls guide.

1. Make sure your subscription has enough LINK to pay for your requests. Also,

you must maintain a minimum balance to upload encrypted secrets to the DON (Read the minimum balance for uploading encrypted secrets section to learn more). You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager. If your subscription runs out of LINK, follow the Fund a Subscription guide. This guide recommends maintaining at least 2 LINK within your subscription.

1. Get a free API key from CoinMarketCap and note your API key.

1. Run `npx env-enc set` to add an encrypted `COINMARKETCAPAPIKEY` to your `.env.enc` file.

```
shell
npx env-enc set
```

## Deploy an Automated Functions Consumer contract

The consumer contract for Automated Functions is different from the consumer in other tutorials. Deploy the `AutomatedFunctionsConsumerExample` contract on Ethereum Sepolia:

1. Open the `AutomatedFunctionsConsumerExample.sol` in Remix.

```
<CodeSample
src="samples/ChainlinkFunctions/AutomatedFunctionsConsumerExample.sol"
showButtonOnly={true} />
```

1. Compile the contract.

1. Open MetaMask and select the Ethereum Sepolia network.

1. In Remix under the Deploy & Run Transactions tab, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Ethereum Sepolia.

1. Under the Deploy section, fill in the router address for your specific blockchain. You can find this address on the Supported Networks page. For Ethereum Sepolia, the router address is `<CopyText text="0xb83E47C2bC239B3bf370bc41e1459A34b41238D0" code/>`.

1. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Ethereum Sepolia.

1. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy your contract address.

Add your Consumer contract to your Functions subscription

Add your contract as an approved consumer contract to your Functions subscription using the Chainlink Functions Subscription Manager.

## Configure Chainlink Automation

The consumer contract that you deployed is designed to be used with a time-based automation. Follow the instructions in the Automation Job Scheduler guide to register your deployed contract using the Chainlink Automation App. Use the following upkeep settings:

- Trigger: Time-based
- Target contract address: The address of the automated Functions consumer contract that you deployed
- ABI: copy/paste the ABI from `automatedFunctions.json`
- Target function: `sendRequestCBOR`
- Time interval: Every 15 minutes
- Name: Give your upkeep a name
- Gas limit: `<CopyText text="10000000" code/>`
- Starting balance (LINK): `<CopyText text="1" code/>`

You can leave the other settings at their default values for the example in this tutorial.

Note: After creation, check your upkeep details and note the address of the upkeep contract. The upkeep contract is responsible for calling your Functions consumer contract at regular times intervals.

```
<ClickToZoom
 src="/images/chainlink-functions/tutorials/automation/automation-register-
scheduler.jpg"
 alt="Register Functions consumer with automation scheduler"
/>
```

<Aside type="note" title="Monitor your balances">

There are two balances that you must monitor:

- Your subscription balance: Your balance will be charged each time your Chainlink Functions is fulfilled. If your balance is insufficient, your contract cannot send requests. Automating your Chainlink Functions means they will be regularly triggered, so monitor and fund your subscription account regularly. You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager.

- Your upkeep balance: You can check this balance on the Chainlink Automation App. The upkeep balance pays Chainlink Automation Network to send your requests according to your provided time interval. Chainlink Automation will not trigger your requests if your upkeep balance runs low.

</Aside>

Configure your Consumer contract

Two important steps are done here:

1. Configure your contract so only the upkeep contract can call the `sendRequestCBOR` function. This security measure is important to prevent anyone from calling several times `sendRequestCBOR` and draining your Functions subscription balance. Follow these steps:

1. On RemixIDE, under the Deploy & Transactions tab, locate your deployed Functions consumer contract.

1. Open the list of functions.

1. Fill in the `setAutomationCronContract` function with the upkeep contract address you copied from the previous step.

1. Click on `transact`. A Metamask popup appears and asks you to confirm the transaction.

1. Confirm the transaction and wait for it to be confirmed.

1. Configure the request details by calling the `updateRequest` function. This step stores the encoded request (source code, reference to encrypted secrets if any, arguments), gas limit, subscription ID, and job ID in the contract storage (see Examine the code). To do so, follow these steps:

1. On a terminal, change directories to the `10-automate-functions` directory.

1. Open `updateRequest.js` and replace the consumer contract address and the subscription ID with your own values:

```
javascript
const consumerAddress = "0x5abE77Ba2aE8918bfd96e2e382d5f213f10D39fA" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

1. Run the `updateRequest.js` script.



```
javascript
const consumerAddress = "0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA" //
REPLACE this with your Functions consumer address
```

1. Run the readLatest script.

```
shell
node examples/10-automate-functions/readLatest.js
```

Output example:

```
text
$ node examples/10-automate-functions/readLatest.js
secp256k1 unavailable, reverting to browser version
Last request ID is
0xa6c0a45c9a24981e0112381f9addeeb8f8a9ad0ea91dd0426703eaa11d5a773a
â€¦ Decoded response to uint256: 6675568n
```

Clean up

After you finish the guide, cancel your upkeep in the Chainlink Automation App and withdraw the remaining funds. After you cancel the upkeep, there is a 50-block delay before you can withdraw the funds.

Examine the code

AutomatedFunctionsConsumer.sol

```
<ChainlinkFunctions section="automated-functions-consumer" />
```

source.js

The JavaScript code is similar to the Call Multiple Data Sources tutorial.

updateRequest.js

This explanation focuses on the updateRequest.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to encode a request offchain and store it in your contract. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/automatedFunctions.json: The ABI of the contract your script will interact with. Note: The script was tested with this AutomatedFunctionsConsumer contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is `updateRequestSepolia`. This function consists of five main parts:

#### 1. Definition of necessary identifiers:

- `routerAddress`: Chainlink Functions router address on the Sepolia testnet.
- `donId`: Identifier of the DON that will fulfill your requests on the Sepolia testnet.
- `gatewayUrls`: The secrets endpoint URL to which you will upload the encrypted secrets.
- `explorerUrl`: Block explorer URL of the Sepolia testnet.
- `source`: The source code must be a string object. That's why we use `fs.readFileSync` to read `source.js` and then call `toString()` to get the content as a string object.
- `args`: During the execution of your function, These arguments are passed to the source code. The `args` value is `["1", "bitcoin", "btc-bitcoin"]`. These arguments are BTC IDs at CoinMarketCap, CoinGecko, and Coinpaprika. You can adapt `args` to fetch other asset prices.
- `secrets`: The secrets object that will be encrypted.
- `slotIdNumber`: Slot ID at the DON where to upload the encrypted secrets.
- `expirationTimeMinutes`: Expiration time in minutes of the encrypted secrets.
- `gasLimit`: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

#### 1. Simulating your request in a local sandbox environment:

- Use `simulateScript` from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

#### 1. Encrypt the secrets, upload the encrypted secrets to the DON, and then encode the reference to the DON-hosted encrypted secrets. This is done in three steps:

- Initialize a `SecretsManager` instance from the Functions NPM package, then call the `encryptSecrets` function.
- Call the `uploadEncryptedSecretsToDON` function of the `SecretsManager` instance. This function returns an object containing a success boolean as long as version, the secret version on the DON storage.
- Call the `buildDONHostedEncryptedSecretsReference` function of the `SecretsManager` instance and use the slot ID and version to encode the DON-hosted encrypted secrets reference.

#### 1. Encode the request data offchain using the `buildRequestCBOR` function from the Functions NPM package.

#### 1. Update the Functions consumer contract:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `updateRequest` function of your consumer contract.

`readLatest.js`

This explanation focuses on the `readLatest.js` script that reads your consumer contract's latest received response and decodes it offchain using the Chainlink Function NPM package.

The script contains a hardcoded value that you must replace with your own Functions consumer contract address:

```
javascript
const consumerAddress = "0x5abE77Ba2aE8918bfd96e2e382d5f213f10D39fA" // REPLACE
this with your Functions consumer address
```

The primary function that the script executes is `readLatest`. This function can be broken down into two main parts:

1. Read the latest response:

- Initialize your functions consumer contract using the contract address, ABI, and ethers provider.
- Call the `slastRequestId`, `slastResponse`, and `slastError` functions of your consumer contract.

1. Decode the latest response:

- If there was an error, read the latest error and parse it into a string.
- If there was no error, use the Functions NPM package's `decodeResult` function and the `ReturnType` enum to decode the response to the expected return type (`ReturnType.uint256` in this example).

```
encode-request-offchain.mdx:
```

```

section: chainlinkFunctions
date: Last Modified
title: "Encode request data offchain"

```

```
import { Aside } from "@components"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how make multiple API calls from your smart contract to a Decentralized Oracle Network. After OCR completes offchain computation and aggregation, the DON returns the asset price to your smart contract. This example returns the BTC/USD price.

This guide assumes that you know how to build HTTP requests and how to use secrets. Read the API query parameters and API use secrets guides before you follow the example in this document.

To build a decentralized asset price, send a request to the DON to fetch the price from many different API providers. Then, calculate the median price. The API providers in this example are:

- CoinMarket
- CoinGecko
- CoinPaprika

Read the Call Multiple Data Sources tutorial before you follow the steps in this example. This tutorial uses the same example but with a slightly different process:

- Instead of sending the request data (source code, encrypted secrets reference, and arguments) in the request, you will first encode it offchain and then send the encoded request. Encoding the request offchain from your front end or a server rather than onchain from your smart contract. This helps save gas.

```
<Aside type="caution">
```

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not



operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possibly revealing of the secret as new versions are released, or other issues.

</Aside>

<Aside type="note">

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

</Aside>

<ChainlinkFunctions section="prerequisites-guides" />

## Tutorial

This tutorial is configured to get the median BTC/USD price from multiple data sources. For a detailed explanation of the code example, read the Examine the code section.

You can locate the scripts used in this tutorial in the examples/9-send-cbor directory.

1. Make sure your subscription has enough LINK to pay for your requests. Also, you must maintain a minimum balance to upload encrypted secrets to the DON (Read the minimum balance for uploading encrypted secrets section to learn more). You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager. If your subscription runs out of LINK, follow the Fund a Subscription guide. This guide recommends maintaining at least 2 LINK within your subscription.

1. Get a free API key from CoinMarketCap and note your API key.

1. Run `npx env-enc set` to add an encrypted `COINMARKETCAPAPIKEY` to your `.env.enc` file.

```
shell
npx env-enc set
```

To run the example:

1. Open the file `request.js`, which is located in the `9-send-cbor` folder.  
1. Replace the consumer contract address and the subscription ID with your own values.

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

1. Make a request:

```
shell
node examples/9-send-cbor/request.js
```



0.236443904180225775 LINK.

- The consumer contract received a response in bytes with a value of 0x0000000000000000000000000000000000000000000000000000000000653c78. Decoding it offchain to uint256 gives you a result: 6634616. The median BTC price is 66346.16 USD.

Examine the code

FunctionsConsumerExample.sol

```
<ChainlinkFunctions section="functions-consumer" />
```

## JavaScript example

source.js

The JavaScript code is similar to the [Call Multiple Data Sources](#) tutorial.

```
request.js
```

This explanation focuses on the `request.js` script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsClient.json: The abi of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerExample contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBED20A71397064D9" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is `makeRequestSepolia`. This function can be broken into seven main parts:

### 1. Definition of necessary identifiers:

- routerAddress: Chainlink Functions router address on the Sepolia testnet.
- donId: Identifier of the DON that will fulfill your requests on the Sepolia testnet.
- gatewayUrls: The secrets endpoint URL to which you will upload the encrypted secrets.
- explorerUrl: Block explorer url of the Sepolia testnet.
- source: The source code must be a string object. That's why we use `fs.readFileSync` to read `source.js` and then call `toString()` to get the content as a string object.
- args: During the execution of your function, These arguments are passed to the source code. The args value is `["1", "bitcoin", "btc-bitcoin"]`. These arguments are BTC IDs at CoinMarketCap, CoinGecko, and Coinpaprika. You can adapt args to fetch other asset prices.
- secrets: The secrets object that will be encrypted.

- slotIdNumber: Slot ID at the DON where to upload the encrypted secrets.
- expirationTimeMinutes: Expiration time in minutes of the encrypted secrets.
- gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

#### 1. Simulating your request in a local sandbox environment:

- Use simulateScript from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package decodeResult function and ReturnType enum to decode the response to the expected returned type (ReturnType.uint256 in this example).

#### 1. Estimating the costs:

- Initialize a SubscriptionManager from the Functions NPM package, then call the estimateFunctionsRequestCost function.
- The response is returned in Juels (1 LINK = 10<sup>18</sup> Juels). Use the ethers.utils.formatEther utility function to convert the output to LINK.

#### 1. Encrypt the secrets, upload the encrypted secrets to the DON, and then encode the reference to the DON-hosted encrypted secrets. This is done in three steps:

- Initialize a SecretsManager instance from the Functions NPM package, then call the encryptSecrets function.
- Call the uploadEncryptedSecretsToDON function of the SecretsManager instance. This function returns an object containing a success boolean as long as version, the secret version on the DON storage.
- Call the buildDONHostedEncryptedSecretsReference function of the SecretsManager instance and use the slot ID and version to encode the DON-hosted encrypted secrets reference.

#### 1. Encode the request data offchain using the buildRequestCBOR function from the Functions NPM package.

#### 1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Make a static call to the sendRequestCBOR function of your consumer contract to return the request ID that Chainlink Functions will generate.
- Call the sendRequestCBOR function of your consumer contract. Note: The encoded data that was generated by buildRequestCBOR is passed in the request.

#### 1. Waiting for the response:

- Initialize a ResponseListener from the Functions NPM package and then call the listenForResponseFromTransaction function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package decodeResult function and ReturnType enum to decode the response to the expected returned type (ReturnType.uint256 in this example).

# importing-packages.mdx:

```

section: chainlinkFunctions
date: Last Modified
title: "Using Imports with Functions"
metadata:
 linkToWallet: true
```

---

```
import { Aside, CopyText } from "@components"
import { Tabs } from "@components/Tabs"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial demonstrates how to import modules and use them with your Functions source code. Modules that are imported into Functions source code must meet the following requirements:

- Each import must be 10 MB or less in size.
- Up to 100 imports total are supported.
- Deno supports ESM compatible NPM imports and some standard Node modules. See the Compatibility List for details.
- Third-party modules are imported at runtime, so import statements must use asynchronous logic like the following examples:

- Importing from deno.land:

```
javascript
const { escape } = await import("https://deno.land/std/regexp/mod.ts")
```

- ESM-compatible packages:

```
javascript
const { format } = await import("npm:date-fns")
```

- Standard Node modules:

```
javascript
const path = await import("node:path")
```

- CDN imports:

```
javascript
const lodash = await import("http://cdn.skypack.dev/lodash")
```

- Imported modules abide by all sandbox restrictions and do not have access to the file system, environment variables, or any other Deno permissions.

<Aside type="caution">

Users are fully responsible for any dependencies their JavaScript source code imports. Chainlink is not responsible

for any imported dependencies and provides no guarantees of the validity, availability or security of any libraries a

user chooses to import or the repositories from which these dependencies are downloaded. Developers are advised to

fully vet any imported dependencies or avoid dependencies altogether to avoid any risks associated with a compromised

library or a compromised repository from which the dependency is downloaded.

</Aside>

<ChainlinkFunctions section="prerequisites-guides" />

## Tutorial

This example imports ethers and demonstrates how to call a smart contract functions using a JSON RPC provider to call an onchain function. In this example, the source code calls the latestRoundData() function from the

AggregatorV3Interface. Read the [Examine the code section](#) for a detailed description of the code example.

You can locate the scripts used in this tutorial in the `examples/11-package-imports` directory.

To run the example:

1. Open the file `request.js`, which is located in the `11-package-imports` folder.
1. Replace the consumer contract address and the subscription ID with your own values.

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

1. Make a request:

```
shell
node examples/11-package-imports/request.js
```

The script runs your function in a sandbox environment before making an onchain transaction:

```
text
$ node examples/11-package-imports/request.js
secp256k1 unavailable, reverting to browser version
Start simulation...
Simulation result {
 capturedTerminalOutput: 'Fetched BTC / USD price: 6644228390483\n',
 responseBytesHexString:
'0x0060afadf7e53'
}
â€¦ Decoded response to int256: 6644228390483n

Estimate request costs...
Fulfillment cost estimated to 1.09518769822223 LINK

Make request...

â€¦ Functions request sent! Transaction hash
0xa73d895adb28360d1737a695647390a3a7f000368d976135fcfe9c834ee75ed6. Waiting for
a response...
See your request in the explorer
https://sepolia.etherscan.io/tx/0xa73d895adb28360d1737a695647390a3a7f000368d9761
35fcfe9c834ee75ed6

â€¦ Request
0x5bac800974596113a3013bf788919a6b3df5ea65ec6238a1c98114a60daff6d2 successfully
fulfilled. Cost is 0.236901088749168095 LINK.Complete reponse: {
 requestId:
'0x5bac800974596113a3013bf788919a6b3df5ea65ec6238a1c98114a60daff6d2',
 subscriptionId: 2303,
 totalCostInJuels: 236901088749168095n,
 responseBytesHexString:
'0x0060afadf7e53',
 errorString: '',
 returnDataBytesHexString: '0x',
 fulfillmentCode: 0
}
```

```
âœœ... Decoded response to int256: 6644228390483n
```

The output of the example gives you the following information:

- [illegible]

Examine the code

FunctionsConsumerExample.sol

```
<ChainlinkFunctions section="functions-consumer" />
```

## JavaScript example

source.js

The Decentralized Oracle Network will run the JavaScript code. The code is self-explanatory and has comments to help you understand all the steps.

```
<ChainlinkFunctions section="deno-importe-notes" />
```

The example `source.js` file uses a JSON RPC call to the `latestRoundData()` function of a Chainlink Data Feed.

The request requires a few modifications to work in the Chainlink Functions environment. For example, the `JsonRpcProvider` class must be inherited to override the `JsonRpcProvider` `send` method. This customization is necessary because Deno does not natively support Node.js modules like `http` or `https`. We override the `send` method to use the `fetch` API, which is the standard way to make HTTP(s) requests in Deno. Note: The `url` passed in the constructor is the URL of the JSON RPC provider.

```

javascript
// Chainlink Functions compatible Ethers JSON RPC provider class
// (this is required for making Ethers RPC calls with Chainlink Functions)
class FunctionsJsonRpcProvider extends ethers.JsonRpcProvider {
 constructor(url) {
 super(url)
 this.url = url
 }
 async send(payload) {
 let resp = await fetch(this.url, {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify(payload),
 })
 return resp.json()
 }
}

```

After the class is extended, you can initialize the provider object with the RPCURL and await the response.

```
javascript
const provider = new FunctionsJsonRpcProvider(RPCURL)
const dataFeedContract = new ethers.Contract(CONTRACTADDRESS, abi, provider)
const dataFeedResponse = await dataFeedContract.latestRoundData()
```

In this example, the contract returns an int256 value. Encode the value so request.js can properly decode it.

```
javascript
return Functions.encodeInt256(dataFeedResponse.answer)
```

request.js

This explanation focuses on the request.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsClient.json: The abi of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerExample contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x91257aa1c6b7f382759c357fbc53c565c80f7fee" // REPLACE
this with your Functions consumer address
const subscriptionId = 38 // REPLACE this with your subscription ID
```

The primary function that the script executes is makeRequestSepolia. This function consists of five main parts:

#### 1. Definition of necessary identifiers:

- routerAddress: Chainlink Functions router address on the Sepolia testnet.
- donId: Identifier of the DON that will fulfill your requests on the Sepolia testnet.
- explorerUrl: Block explorer URL of the Sepolia testnet.
- source: The source code must be a string object. That's why we use fs.readFileSync to read source.js and then call toString() to get the content as a string object.
- args: During the execution of your function, these arguments are passed to the source code.
- gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.



## 1. Simulating your request in a local sandbox environment:

- Use `simulateScript` from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.int256` in this example).

## 1. Estimating the costs:

- Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost`.
- The response is returned in Juels (1 LINK = 10<sup>18</sup> Juels). Use the `ethers.utils.formatEther` utility function to convert the output to LINK.

## 1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `sendRequest` function of your consumer contract.

## 1. Waiting for the response:

- Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.int256` in this example).

# index.mdx:

---

```
section: chainlinkFunctions
date: Last Modified
title: "Chainlink Functions Tutorials"
isIndex: true

```

```
import { Aside } from "@components"
```

```
<Aside type="note" title="Explore Chainlink Functions Playground">
 Start exploring Chainlink Functions in your browser using the Chainlink
 Functions
 Playground. You can use the playground to simulate Chainlink Functions, call
 APIs, and execute demo requests.
</Aside>
```

## Topics

- Simple Computation
- Call an API
- POST Data to an API
- Using DON-hosted Secrets in Requests
- Using Imports with Functions
- Use ABI encoding and decoding
- Using User-hosted (gist) Secrets in Requests
- Using User-hosted Secrets in Requests
- Call Multiple Data Sources
- Encode request data offchain
- Automate your Functions (Time-based Automation)
- Automate your Functions (Custom Logic Automation)

```
simple-computation.mdx:
```

```

section: chainlinkFunctions
date: Last Modified
title: "Request Computation"
metadata:
 linkToWallet: true
whatsnext:
 {
 "Try out the Chainlink Functions Tutorials":
 "/chainlink-functions/tutorials",
 "Read the Architecture to understand how Chainlink Functions operates":
 "/chainlink-functions/resources/architecture",
 }

```

```
import { Aside, CopyText } from "@components"
import { Tabs } from "@components/Tabs"
import ChainlinkFunctions from
"@features/chainlink-functions/common/ChainlinkFunctions.astro"
```

This tutorial shows you how to run computations on the Chainlink Functions Decentralized Oracle Network (DON). The example code computes the geometric mean of numbers in a list. After OCR completes offchain computation and aggregation, it returns the result to your smart contract.

```
<ChainlinkFunctions section="prerequisites-guides" />
```

## Tutorial

This tutorial is configured to get the average (geometric mean) from a list of numbers 1,2,3,4,5,6,7,8,9,10. Read the Examine the code section for a detailed description of the code example.

You can locate the scripts used in this tutorial in the examples/1-simple-computation directory.

To run the example:

1. Open the file request.js, which is located in the 1-simple-computation folder.
1. Replace the consumer contract address and the subscription ID with your own values.

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" //
REPLACE this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

1. Make a request:

```
shell
node examples/1-simple-computation/request.js
```

The script runs your function in a sandbox environment before making an onchain transaction:

```
text
$ node examples/1-simple-computation/request.js

secp256k1 unavailable, reverting to browser version
```



The Decentralized Oracle Network will run the JavaScript code. The code is self-explanatory and has comments to help you understand all the steps. Note: Functions requests with custom source code can use vanilla Deno but cannot use any require statements. Import statements and imported modules are supported only on testnets.

The main steps are:

- Read the numbers provided as arguments in the args setting. Because args is an array of string, call parseInt to convert from string to number. Note: args contains string values that are injected into the JavaScript source code when the Decentralized Oracle Network executes your function. You can access these values from your JavaScript code using the name args.
- Calculate the average (geometric mean): First, compute the product of the numbers. Then, calculate the nth root of the product where n is the length of args.
- Return the result as a buffer using the Functions.encodeUint256 helper function. Because Solidity doesn't support decimals, multiply the result by 100 and round the result to the nearest integer. There are other helper functions that you could use depending on the response type:
  - Functions.encodeUint256: Takes a positive JavaScript integer number and returns a Buffer of 32 bytes representing a uint256 type in Solidity.
  - Functions.encodeInt256: Takes a JavaScript integer number and returns a Buffer of 32 bytes representing an int256 type in Solidity.
  - Functions.encodeString: Takes a JavaScript string and returns a Buffer representing a string type in Solidity.

Note: You are not required to use these encoding functions as long as the JavaScript code returns a Buffer representing the bytes array returned onchain. Read this article if you are new to Javascript Buffers and want to understand why they are important.

request.js

This explanation focuses on the request.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path and fs : Used to read the source file.
- ethers: Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in the NPM README.
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read the official documentation to learn more.
- ../abi/functionsClient.json: The abi of the contract your script will interact with. Note: The script was tested with this FunctionsConsumerExample contract.

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
javascript
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9" // REPLACE
this with your Functions consumer address
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is makeRequestSepolia. This function consists of five main parts:

## 1. Definition of necessary identifiers:

- routerAddress: Chainlink Functions router address on the Sepolia testnet.
- donId: Identifier of the DON that will fulfill your requests on the Sepolia testnet.
- explorerUrl: Block explorer URL of the Sepolia testnet.
- source: The source code must be a string object. That's why we use `fs.readFileSync` to read `source.js` and then call `toString()` to get the content as a string object.
- args: During the execution of your function, These arguments are passed to the source code.
- gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- Initialization of ethers signer and provider objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.

## 1. Simulating your request in a local sandbox environment:

- Use `simulateScript` from the Chainlink Functions NPM package.
- Read the response of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

## 1. Estimating the costs:

- Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost`.
- The response is returned in Juels (1 LINK = 10<sup>18</sup> Juels). Use the `ethers.utils.formatEther` utility function to convert the output to LINK.

## 1. Making a Chainlink Functions request:

- Initialize your functions consumer contract using the contract address, abi, and ethers signer.
- Call the `sendRequest` function of your consumer contract.

## 1. Waiting for the response:

- Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
- Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

# index.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Chainlink Local"
isIndex: true

```

```
import { YouTube } from "@astro-community/astro-embed-youtube"
```

Chainlink Local is an installable package that allows you to run Chainlink services locally. You can import Chainlink Local into your preferred local development environment, such as Foundry projects, Hardhat scripts, or the Remix IDE. Chainlink Local enables rapid exploration, prototyping, local development, and iteration with Chainlink services

before transitioning to a testnet.

For instance, you can use Chainlink Local to execute CCIP token transfers and arbitrary messages on a local Hardhat or Anvil (Foundry) development node. Chainlink Local also supports forked nodes, allowing you to work with multiple locally running blockchain networks using historical network states. User contracts tested with Chainlink Local can be deployed to test networks without modifications, ensuring a seamless transition from local development to live testnets.

#### Key Features of Chainlink Local:

- Local Simulation: Run Chainlink services on a local development blockchain node, enabling fast and efficient testing and prototyping.
- Forked Networks: Work with deployed Chainlink contracts using one or multiple forked networks, providing a more realistic testing environment.
- Seamless Integration: Integrate with Foundry, Hardhat, and Remix IDE for a streamlined development process.

To get started testing CCIP with Chainlink Local, follow the installation and setup steps in the CCIP guides for Foundry, Hardhat, or Remix IDE.

<YouTube id="https://www.youtube.com/watch?v=rEVjU9tOf74" />

# index.mdx:

---

section: chainlinkLocal  
date: Last Modified  
title: "Chainlink Local API reference"  
isIndex: true

---

import Common from "@features/chainlink-local/Common.astro"

<Common callout="importPackage" />

- JavaScript
- Solidity

# CCIPLocalSimulatorFork.mdx:

---

section: chainlinkLocal  
date: Last Modified  
title: "CCIPLocalSimulatorFork API reference"  
isIndex: true

---

#### Typedefs

```
<dl>
 <dt>
 Evm2EvmMessage : <code>Object</code>
 </dt>
 <dd></dd>
</dl>
```

<a name="requestLinkFromTheFaucet"></a>

requestLinkFromTheFaucet(linkAddress, to, amount) â†’  
<code>Promise.&lt;string></code>

Requests LINK tokens from the faucet and returns the transaction hash

Kind: global function

Returns: `Promise.<string>` - Promise resolving to the transaction hash of the fund transfer

Param	Type	Description
-----	-----	-----
linkAddress	<code>string</code>	The address of the LINK contract on the current network
to	<code>string</code>	The address to send LINK to
amount	<code>bigint</code>	The amount of LINK to request

</a>

`getEvm2EvmMessage(receipt) â†’ <code>Evm2EvmMessage</code> \ | <code>null</code>`

Parses a transaction receipt to extract the sent message Scans through transaction logs to find a CCIPSendRequested event and then decodes it to an object

Kind: global function

Returns: `Evm2EvmMessage` \ |

`null` - Returns either the sent message or null if provided receipt does not contain CCIPSendRequested log

Param	Type	Description
-----	-----	-----
receipt	<code>object</code>	The transaction receipt from the ccipSend call

</a>

`routeMessage(routerAddress, evm2EvmMessage) â†’ <code>Promise.<void></code>`

Routes the sent message from the source network on the destination (current) network

Kind: global function

Returns: `Promise.<void>` - Either resolves with no value if the message is successfully routed, or reverts

Throws:

- `Error` Fails if no off-ramp matches the message's source chain selector or if calling `router.getOffRamps()`

Param	Type	Description
-----	-----	-----
routerAddress	<code>string</code>	Address of the destination Router
evm2EvmMessage	<code>Evm2EvmMessage</code>	Sent cross-chain message

<a name="Evm2EvmMessage"></a>

Evm2EvmMessage : <code>Object</code>

Kind: global typedef

Properties

Name	Type
-----	
sourceChainSelector	<code>bigint</code>
sender	<code>string</code>
receiver	<code>string</code>
sequenceNumber	<code>bigint</code>
gasLimit	<code>bigint</code>
strict	<code>boolean</code>
nonce	<code>bigint</code>
feeToken	<code>string</code>
feeTokenAmount	<code>bigint</code>
data	<code>string</code>
tokenAmounts	<code>Array.&lt;\{token: string, amount: bigint}&gt;</code>
sourceTokenData	<code>Array.&lt;string&gt;</code>
messageId	<code>string</code>

# index.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Chainlink Local JavaScript API reference"
isIndex: true

```

- CCIPLocalSimulatorFork

# index.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Chainlink Local Solidity contracts API reference"
isIndex: true

```

- ccip
- data-feeds
- shared



# BurnMintERC677Helper.mdx:

---

section: chainlinkLocal  
date: Last Modified  
title: "BurnMintERC677Helper API reference"  
isIndex: true

---

## BurnMintERC677Helper

This contract extends the functionality of the BurnMintERC677 token contract to include a drip function that mints one full token to a specified address.

Inherits from the BurnMintERC677 contract and sets the token name, symbol, decimals, and initial supply in the constructor.

### constructor

```
solidity
constructor(string name, string symbol) public
```

Constructor to initialize the BurnMintERC677Helper contract with a name and symbol.

Calls the parent constructor of BurnMintERC677 with fixed decimals (18) and initial supply (0).

### Parameters

Name	Type	Description
name	string	The name of the token.
symbol	string	The symbol of the token.

### drip

```
solidity
function drip(address to) external
```

Mints one full token (1e18) to the specified address.

Calls the internal mint function from the BurnMintERC677 contract.

### Parameters

Name	Type	Description
to	address	The address that receives the minted token.

# CCIPLocalSimulator.mdx:

---

section: chainlinkLocal  
date: Last Modified  
title: "CCIPLocalSimulator API reference"  
isIndex: true

---

## CCIPLocalSimulator

This contract simulates local CCIP (Cross-Chain Interoperability Protocol) operations for testing and development purposes.

This contract includes methods to manage supported tokens and configurations for local simulations.

CHAINSELECTOR

```
solidity
uint64 CHAINSELECTOR
```

The unique CCIP Chain Selector constant

iwrappedNative

```
solidity
contract WETH9 iwrappedNative
```

The wrapped native token instance

ilinkToken

```
solidity
contract LinkToken ilinkToken
```

The LINK token instance

iccipBnM

```
solidity
contract BurnMintERC677Helper iccipBnM
```

The BurnMintERC677Helper instance for CCIP-BnM token

iccipLnM

```
solidity
contract BurnMintERC677Helper iccipLnM
```

The BurnMintERC677Helper instance for CCIP-LnM token

imockRouter

```
solidity
contract MockCCIPRouter imockRouter
```

The mock CCIP router instance

ssupportedTokens

```
solidity
address[] ssupportedTokens
```

The list of supported token addresses

constructor

```
solidity
constructor() public
```

Constructor to initialize the contract and pre-deployed token instances

supportNewToken

```
solidity
function supportNewToken(address tokenAddress) external
```

Allows user to support any new token, besides CCIP BnM and CCIP LnM, for cross-chain transfers.

Parameters

Name	Type	Description
tokenAddress	address	The address of the token to add to the list of supported tokens.

isChainSupported

```
solidity
function isChainSupported(uint64 chainSelector) public pure returns (bool supported)
```

Checks whether the provided chainSelector is supported by the simulator.

Parameters

Name	Type	Description
chainSelector	uint64	The unique CCIP Chain Selector.

Return Values

Name	Type	Description
supported	bool	Returns true if chainSelector is supported by the simulator.

getSupportedTokens

```
solidity
function getSupportedTokens(uint64 chainSelector) external view returns (address[] tokens)
```

Gets a list of token addresses that are supported for cross-chain transfers by the simulator.

Parameters

Name	Type	Description
chainSelector	uint64	The unique CCIP Chain Selector.

## Return Values

Name	Type	Description
-----	-----	-----

-----  
| tokens | address[] | Returns a list of token addresses that are supported for cross-chain transfers by the simulator. |

requestLinkFromFaucet

solidity

function requestLinkFromFaucet(address to, uint256 amount) external returns (bool success)

Requests LINK tokens from the faucet. The provided amount of tokens are transferred to provided destination address.

## Parameters

Name	Type	Description
-----	-----	-----
to	address	The address to which LINK tokens are to be sent.
amount	uint256	The amount of LINK tokens to send.

## Return Values

Name	Type	Description
-----	-----	-----

-----  
| success | bool | Returns true if the transfer of tokens was successful, otherwise false. |

configuration

solidity

function configuration() public view returns (uint64 chainSelector, contract IRouterClient sourceRouter, contract IRouterClient destinationRouter, contract WETH9 wrappedNative, contract LinkToken linkToken, contract BurnMintERC677Helper ccipBnM, contract BurnMintERC677Helper ccipLnM)

Returns configuration details for pre-deployed contracts and services needed for local CCIP simulations.

## Return Values

Name	Type	Description
-----	-----	-----
chainSelector\Selector.	uint64	The unique CCIP Chain
sourceRouter\contract.	contract IRouterClient	The source chain Router
destinationRouter\Router contract.	contract IRouterClient	The destination chain
wrappedNative\which can be used for CCIP fees.	contract WETH9	The wrapped native token
linkToken\	contract LinkToken	The LINK token.

ccipBnM\	contract BurnMintERC677Helper   The ccipBnM token.
ccipLnM\	contract BurnMintERC677Helper   The ccipLnM token.

# CCIPLocalSimulatorFork.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "CCIPLocalSimulatorFork API reference"
isIndex: true

```

IRouterFork

OffRamp

```

solidity
struct OffRamp {
 uint64 sourceChainSelector;
 address offRamp;
}
```

getOffRamps

```

solidity
function getOffRamps() external view returns (struct IRouterFork.OffRamp[])
```

Gets the list of offRamps

Return Values

Name	Type	Description
[0]	struct IRouterFork.OffRamp[]	offRamps - Array of OffRamp structs

IEVM2EVMOffRampFork

executeSingleMessage

```

solidity
function executeSingleMessage(struct Internal.EVM2EVMMessage message, bytes[]
offchainTokenData) external
```

Executes a single CCIP message on the offRamp

Parameters

Name	Type	Description
message	struct Internal.EVM2EVMMessage	The CCIP message to be executed
offchainTokenData	bytes[]	Additional offchain token data

CCIPLocalSimulatorFork

Works with Foundry only

CCIPSendRequested

```
solidity
event CCIPSendRequested(struct Internal.EVM2EVMMessage message)
```

Event emitted when a CCIP send request is made

Parameters

Name	Type	Description
message	struct Internal.EVM2EVMMessage	The EVM2EVM message that was sent

iregister

```
solidity
contract Register iregister
```

The immutable register instance

LINKFAUCET

```
solidity
address LINKFAUCET
```

The address of the LINK faucet

sprocessedMessages

```
solidity
mapping(bytes32 => bool) sprocessedMessages
```

Mapping to track processed messages

constructor

```
solidity
constructor() public
```

Constructor to initialize the contract

switchChainAndRouteMessage

```
solidity
function switchChainAndRouteMessage(uint256 forkId) external
```

To be called after the sending of the cross-chain message (ccipSend). Goes through the list of past logs and looks for the CCIPSendRequested event. Switches to a destination network fork. Routes the sent cross-chain message on the destination network.

Parameters

Name	Type	Description

Name	Type	Description
forkId	uint256	The ID of the destination network fork. This is the returned value of createFork() or createSelectFork()

## getNetworkDetails

solidity

```
function getNetworkDetails(uint256 chainId) external view returns (struct Register.NetworkDetails)
```

Returns the default values for currently CCIP supported networks. If network is not present or some of the values are changed, user can manually add new network details using the setNetworkDetails function.

## Parameters

Name	Type	Description
chainId	uint256	The blockchain network chain ID. For example 11155111 for Ethereum Sepolia. Not CCIP chain selector.

## Return Values

Name	Type	Description
[0]	struct Register.NetworkDetails	networkDetails - The tuple containing: chainSelector - The unique CCIP Chain Selector. routerAddress - The address of the CCIP Router contract. linkAddress - The address of the LINK token. wrappedNativeAddress - The address of the wrapped native token that can be used for CCIP fees. ccipBnMAddress - The address of the CCIP BnM token. ccipLnMAddress - The address of the CCIP LnM token.

## setNetworkDetails

solidity

```
function setNetworkDetails(uint256 chainId, struct Register.NetworkDetails networkDetails) external
```

If network details are not present or some of the values are changed, user can manually add new network details using the setNetworkDetails function.

## Parameters

Name	Type	Description
------	------	-------------

chainId	uint256	The blockchain network chain ID. For example 11155111 for Ethereum Sepolia. Not CCIP chain selector.
---------	---------	------------------------------------------------------------------------------------------------------

networkDetails	struct Register.NetworkDetails	The tuple containing: chainSelector - The unique CCIP Chain Selector. routerAddress - The address of the CCIP Router contract. linkAddress - The address of the LINK token. wrappedNativeAddress - The address of the wrapped native token that can be used for CCIP fees. ccipBnMAddress - The address of the CCIP BnM token. ccipLnMAddress - The address of the CCIP LnM token.
----------------	--------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

requestLinkFromFaucet

solidity

function requestLinkFromFaucet(address to, uint256 amount) external returns (bool success)

Requests LINK tokens from the faucet. The provided amount of tokens are transferred to provided destination address.

Parameters

Name	Type	Description
to	address	The address to which LINK tokens are to be sent.
amount	uint256	The amount of LINK tokens to send.

Return Values

Name	Type	Description
success	bool	Returns true if the transfer of tokens was successful, otherwise false.

# index.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Chainlink Local CCIP contracts API reference"
isIndex: true

```

- BurnMintERC677Helper
- CCIPLocalSimulator
- CCIPLocalSimulatorFork
- MockEvm2EvmOffRamp
- Register

# MockEvm2EvmOffRamp.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "MockEvm2EvmOffRamp API reference"
isIndex: true

```

MockEvm2EvmOffRamp



This contract handles off-ramp processes for CCIP messages

DynamicConfig

```
solidity
struct DynamicConfig {
 uint32 permissionLessExecutionThresholdSeconds;
 address router;
 address priceRegistry;
 uint16 maxNumberOfTokensPerMsg;
 uint32 maxDataBytes;
 uint32 maxPoolReleaseOrMintGas;
}
```

isourceChainSelector

```
solidity
uint64 isourceChainSelector
```

chain selector for the source chain

sccipLocalSimulator

```
solidity
address sccipLocalSimulator
```

Address of the CCIP Local Simulator

sdynamicConfig

```
solidity
struct MockEvm2EvmOffRamp.DynamicConfig sdynamicConfig
```

Dynamic configuration of the offramp

CanOnlySimulatorCall

```
solidity
error CanOnlySimulatorCall()
```

Error thrown when a function can only be called by the simulator

ReceiverError

```
solidity
error ReceiverError(bytes error)
```

Error thrown when there is an error in the receiver

Parameters

Name	Type	Description
error	bytes	The error data

TokenHandlingError

```
solidity
```

error TokenHandlingError(bytes error)

Error thrown when there is an error in token handling

Parameters

Name	Type	Description
error	bytes	The error data

UnsupportedToken

solidity

error UnsupportedToken(contract IERC20 token)

Error thrown when an unsupported token is encountered

Parameters

Name	Type	Description
token	contract IERC20	The unsupported token

constructor

solidity

constructor(address ccipLocalSimulator, struct MockEvm2EvmOffRamp.DynamicConfig dynamicConfig, struct RateLimiter.Config config, uint64 sourceChainSelector, address[] sourceTokens, address[] pools) public

Constructor to initialize the contract.

Parameters

Name	Type	Description
ccipLocalSimulator	address	Address of the CCIP local simulator.
dynamicConfig	struct MockEvm2EvmOffRamp.DynamicConfig	Initial dynamic configuration parameters.
config	struct RateLimiter.Config	Rate limiter configuration.
sourceChainSelector	uint64	Source chain selector.
sourceTokens	address[]	List of supported tokens on the source chain.
pools	address[]	List of pools corresponding to the supported tokens on the source chain.

executeSingleMessage

solidity

function executeSingleMessage(struct Internal.EVM2EVMMessage message, bytes[] offchainTokenData) external

Executes a single CCIP message.

Parameters

Name	Type	Description
-----	-----	
message	struct Internal.EVM2EVMMessage	The CCIP message to be executed.
offchainTokenData	bytes[]	Additional off-chain token data.

\releaseOrMintTokens

solidity

```
function releaseOrMintTokens(struct Client.EVMTokenAmount[] sourceTokenAmounts,
bytes originalSender, address receiver, bytes[] sourceTokenData, bytes[]
offchainTokenData) internal returns (struct Client.EVMTokenAmount[])
```

Uses pools to release or mint a number of different tokens to a receiver address.

This function wraps the token pool call in a try-catch block to gracefully handle any non-rate limiting errors that may occur. If we encounter a rate limiting related error we bubble it up. If we encounter a non-rate limiting error we wrap it in a TokenHandlingError.

Parameters

Name	Type	Description
-----	-----	
sourceTokenAmounts	struct Client.EVMTokenAmount[]	List of tokens and amount values to be released/minted.
originalSender	bytes	The message sender.
receiver	address	The address that will receive the tokens.
sourceTokenData	bytes[]	Array of token data returned by token pools on the source chain.
offchainTokenData	bytes[]	Array of token data fetched offchain by the DON.

Return Values

Name	Type	Description
----	-----	
[0]	struct Client.EVMTokenAmount[]	destTokenAmounts - The amounts of tokens released or minted.

getPoolBySourceToken

solidity

```
function getPoolBySourceToken(contract IERC20 sourceToken) public view returns
(contract IPool)
```

Get a token pool by its source token.

Parameters

Name	Type	Description
------	------	-------------

-----	-----	-----
sourceToken	contract IERC20	The source token.

## Return Values

Name	Type	Description
----	-----	-----
[0]	contract IPool	pool - The corresponding token pool.

# Register.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Register API reference"
isIndex: true

```

## Register

This contract allows storing and retrieving network details for various chains.

Stores network details in a mapping based on chain IDs.

### NetworkDetails

```

solidity
struct NetworkDetails {
 uint64 chainSelector;
 address routerAddress;
 address linkAddress;
 address wrappedNativeAddress;
 address ccipBnMAddress;
 address ccipLnMAddress;
}
```

### snetworkDetails

```

solidity
mapping(uint256 => struct Register.NetworkDetails) snetworkDetails
```

Mapping to store network details based on chain ID.

### constructor

```

solidity
constructor() public
```

Constructor to initialize the network details for various chains.

### getNetworkDetails

```

solidity
function getNetworkDetails(uint256 chainId) external view returns (struct
Register.NetworkDetails networkDetails)
```

Retrieves network details for a given chain ID.

### Parameters

Name	Type	Description
chainId	uint256	The ID of the chain to get the details for.

#### Return Values

Name	Type	Description
networkDetails	struct Register.NetworkDetails	The network details for the specified chain ID.

#### setNetworkDetails

solidity

```
function setNetworkDetails(uint256 chainId, struct Register.NetworkDetails
networkDetails) external
```

Sets the network details for a given chain ID.

#### Parameters

Name	Type	Description
chainId	uint256	The ID of the chain to set the details for.
networkDetails	struct Register.NetworkDetails	The network details to set for the specified chain ID.

# index.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Chainlink Local Data feeds API reference"
isIndex: true

```

- MockOffchainAggregator
- MockV3Aggregator
- Data Feeds mock interfaces

# MockOffchainAggregator.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "MockOffchainAggregator API reference"
isIndex: true

```

#### MockOffchainAggregator

This contract is a mock implementation of an offchain aggregator for testing purposes.

This contract simulates the behavior of an offchain aggregator and allows for

updating answers and round data.

decimals

solidity  
uint8 decimals

The number of decimals used by the aggregator.

latestAnswer

solidity  
int256 latestAnswer

The latest answer reported by the aggregator.

latestTimestamp

solidity  
uint256 latestTimestamp

The timestamp of the latest answer.

latestRound

solidity  
uint256 latestRound

The latest round ID.

minAnswer

solidity  
int192 minAnswer

The minimum answer the aggregator is allowed to report.

maxAnswer

solidity  
int192 maxAnswer

The maximum answer the aggregator is allowed to report.

getAnswer

solidity  
mapping(uint256 => int256) getAnswer

Mapping to get the answer for a specific round ID.

getTimestamp

solidity  
mapping(uint256 => uint256) getTimestamp

Mapping to get the timestamp for a specific round ID.

constructor

solidity

```
constructor(uint8 decimals, int256 initialAnswer) public
```

Constructor to initialize the MockOffchainAggregator contract with initial parameters.

Parameters

Name	Type	Description
-----	-----	-----
\decimals	uint8	The number of decimals for the aggregator.
\initialAnswer	int256	The initial answer to be set in the mock aggregator.

updateAnswer

solidity

```
function updateAnswer(int256 answer) public
```

Updates the answer in the mock aggregator.

Parameters

Name	Type	Description
-----	-----	-----
\answer	int256	The new answer to be set.

updateRoundData

solidity

```
function updateRoundData(uint80 roundId, int256 answer, uint256 timestamp, uint256 startedAt) public
```

Updates the round data in the mock aggregator.

Parameters

Name	Type	Description
-----	-----	-----
\roundId	uint80	The round ID to be updated.
\answer	int256	The new answer to be set.
\timestamp	uint256	The timestamp to be set.
\startedAt	uint256	The timestamp when the round started.

getRoundData

solidity

```
function getRoundData(uint80 roundId) external view returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound)
```

Gets the round data for a specific round ID.

Parameters

Name	Type	Description
\roundId	uint80	The round ID to get the data for.

#### Return Values

Name	Type	Description
roundId	uint80	The round ID.
answer	int256	The answer for the round.
startedAt	uint256	The timestamp when the round started.
updatedAt	uint256	The timestamp when the round was updated.
answeredInRound	uint80	The round ID in which the answer was computed.

#### latestRoundData

solidity

function latestRoundData() external view returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound)

Gets the latest round data.

#### Return Values

Name	Type	Description
roundId	uint80	The latest round ID.
answer	int256	The latest answer.
startedAt	uint256	The timestamp when the latest round started.
updatedAt	uint256	The timestamp when the latest round was updated.
answeredInRound	uint80	The round ID in which the latest answer was computed.

#### updateMinAndMaxAnswers

solidity

function updateMinAndMaxAnswers(int192 minAnswer, int192 maxAnswer) external

Updates the minimum and maximum answers the aggregator can report.

#### Parameters

Name	Type	Description
\minAnswer	int192	The new minimum answer.
\maxAnswer	int192	The new maximum answer.

# MockV3Aggregator.mdx:

---

section: chainlinkLocal

date: Last Modified

title: "MockV3Aggregator API reference"

isIndex: true



---

## MockV3Aggregator

This contract is a mock implementation of the AggregatorV2V3Interface for testing purposes.

This contract interacts with a MockOffchainAggregator to simulate price feeds.

### version

solidity  
uint256 version

The version of the aggregator.

### aggregator

solidity  
address aggregator

The address of the current aggregator.

### proposedAggregator

solidity  
address proposedAggregator

The address of the proposed aggregator.

### constructor

solidity  
constructor(uint8 decimals, int256 initialAnswer) public

Constructor to initialize the MockV3Aggregator contract with initial parameters.

### Parameters

Name	Type	Description
-----	-----	
\decimals	uint8	The number of decimals for the aggregator.
\initialAnswer	int256	The initial answer to be set in the mock aggregator.

### decimals

solidity  
function decimals() external view returns (uint8)

Gets the number of decimals used by the aggregator.

### Return Values

Name	Type	Description
----	-----	-----

| [0] | uint8 | uint8 - The number of decimals. |

getAnswer

solidity

function getAnswer(uint256 roundId) external view returns (int256)

Gets the answer for a specific round ID.

Parameters

Name	Type	Description
roundId	uint256	The round ID to get the answer for.

Return Values

Name	Type	Description
[0]	int256	int256 - The answer for the given round ID.

getRoundData

solidity

function getRoundData(uint80 roundId) external view returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound)

Gets the round data for a specific round ID.

This function should raise "No data present" if no data is available for the given round ID.

Parameters

Name	Type	Description
\roundId	uint80	The round ID to get the data for.

Return Values

Name	Type	Description
roundId	uint80	The round ID.
answer	int256	The answer for the round.
startedAt	uint256	The timestamp when the round started.
updatedAt	uint256	The timestamp when the round was updated.
answeredInRound	uint80	The round ID in which the answer was computed.

latestRoundData

solidity

function latestRoundData() external view returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound)

Gets the latest round data.

This function should raise "No data present" if no data is available.

Return Values

Name	Type	Description
------	------	-------------

-----	-----	
roundId	uint80	The latest round ID.
answer	int256	The latest answer.
startedAt	uint256	The timestamp when the latest round started.
updatedAt	uint256	The timestamp when the latest round was updated.
answeredInRound	uint80	The round ID in which the latest answer was computed.

getTimestamp

solidity

function getTimestamp(uint256 roundId) external view returns (uint256)

Gets the timestamp for a specific round ID.

Parameters

Name	Type	Description
-----	-----	-----
roundId	uint256	The round ID to get the timestamp for.

Return Values

Name	Type	Description
----	-----	-----
[0]	uint256	uint256 - The timestamp for the given round ID.

latestAnswer

solidity

function latestAnswer() external view returns (int256)

Gets the latest answer from the aggregator.

Return Values

Name	Type	Description
----	-----	-----
[0]	int256	int256 - The latest answer.

latestTimestamp

solidity

function latestTimestamp() external view returns (uint256)

Gets the timestamp of the latest answer from the aggregator.

Return Values

Name	Type	Description
----	-----	-----
[0]	uint256	uint256 - The timestamp of the latest answer.

latestRound

```
solidity
function latestRound() external view returns (uint256)
```

Gets the latest round ID from the aggregator.

Return Values

Name	Type	Description
[0]	uint256	uint256 - The latest round ID.

updateAnswer

```
solidity
function updateAnswer(int256 answer) public
```

Updates the answer in the mock aggregator.

Parameters

Name	Type	Description
\answer	int256	The new answer to be set.

updateRoundData

```
solidity
function updateRoundData(uint80 roundId, int256 answer, uint256 timestamp,
uint256 startedAt) public
```

Updates the round data in the mock aggregator.

Parameters

Name	Type	Description
\roundId	uint80	The round ID to be updated.
\answer	int256	The new answer to be set.
\timestamp	uint256	The timestamp to be set.
\startedAt	uint256	The timestamp when the round started.

proposeAggregator

```
solidity
function proposeAggregator(contract AggregatorV2V3Interface aggregator) external
```

Proposes a new aggregator.

Parameters

Name	Type	Description
\aggregator	contract AggregatorV2V3Interface	The address of the proposed aggregator.

confirmAggregator

```
solidity
```

function confirmAggregator(address aggregator) external

Confirms the proposed aggregator.

Parameters

Name	Type	Description
\aggregator	address	The address of the proposed aggregator.

description

solidity

function description() external pure returns (string)

Gets the description of the aggregator.

Return Values

Name	Type	Description
[0]	string	string memory - The description of the aggregator.

# AggregatorInterface.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "AggregatorInterface API reference"
isIndex: true

```

AggregatorInterface

Interface for accessing data from an aggregator contract.

Provides methods to get the latest data and historical data for specific rounds.

latestAnswer

solidity

function latestAnswer() external view returns (int256)

Gets the latest answer from the aggregator.

Return Values

Name	Type	Description
[0]	int256	int256 - The latest answer.

latestTimestamp

solidity

function latestTimestamp() external view returns (uint256)

Gets the timestamp of the latest answer from the aggregator.

## Return Values

Name	Type	Description
[0]	uint256	uint256 - The timestamp of the latest answer.

## latestRound

solidity

function latestRound() external view returns (uint256)

Gets the latest round ID from the aggregator.

## Return Values

Name	Type	Description
[0]	uint256	uint256 - The latest round ID.

## getAnswer

solidity

function getAnswer(uint256 roundId) external view returns (int256)

Gets the answer for a specific round ID.

## Parameters

Name	Type	Description
roundId	uint256	The round ID to get the answer for.

## Return Values

Name	Type	Description
[0]	int256	int256 - The answer for the given round ID.

## getTimestamp

solidity

function getTimestamp(uint256 roundId) external view returns (uint256)

Gets the timestamp for a specific round ID.

## Parameters

Name	Type	Description
roundId	uint256	The round ID to get the timestamp for.

## Return Values

Name	Type	Description
[0]	uint256	uint256 - The timestamp for the given round ID.

## AnswerUpdated

solidity

event AnswerUpdated(int256 current, uint256 roundId, uint256 updatedAt)

Emitted when the answer is updated.

#### Parameters

Name	Type	Description
current	int256	The updated answer.
roundId	uint256	The round ID for which the answer was updated.
updatedAt	uint256	The timestamp when the answer was updated.

#### NewRound

solidity

```
event NewRound(uint256 roundId, address startedBy, uint256 startedAt)
```

Emitted when a new round is started.

#### Parameters

Name	Type	Description
roundId	uint256	The round ID of the new round.
startedBy	address	The address of the account that started the round.
startedAt	uint256	The timestamp when the round was started.

# AggregatorV2V3Interface.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "AggregatorV2V3Interface API reference"
isIndex: true

```

#### AggregatorV2V3Interface

Interface that inherits from both  
AggregatorInterface and  
AggregatorV3Interface.

# AggregatorV3Interface.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "AggregatorV3Interface API reference"
isIndex: true

```

#### AggregatorV3Interface

Interface for accessing detailed data from an aggregator contract, including  
round data and metadata.

Provides methods to get the latest data, historical data for specific rounds,  
and metadata such as decimals and description.

decimals

solidity  
function decimals() external view returns (uint8)

Gets the number of decimals used by the aggregator.

Return Values

Name	Type	Description
----	-----	-----
[0]	uint8	uint8 - The number of decimals.

description

solidity  
function description() external view returns (string)

Gets the description of the aggregator.

Return Values

Name	Type	Description
----	-----	-----
[0]	string	string memory - The description of the aggregator.

version

solidity  
function version() external view returns (uint256)

Gets the version of the aggregator.

Return Values

Name	Type	Description
----	-----	-----
[0]	uint256	uint256 - The version of the aggregator.

getRoundData

solidity  
function getRoundData(uint80 roundId) external view returns (uint80 roundId,  
int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound)

Gets the round data for a specific round ID.

This function should raise "No data present" if no data is available for the given round ID.

Parameters

Name	Type	Description
-----	-----	-----
\roundId	uint80	The round ID to get the data for.

Return Values

Name	Type	Description
-----	-----	-----
roundId	uint80	The round ID.
answer	int256	The answer for the round.



startedAt	uint256	The timestamp when the round started.	
updatedAt	uint256	The timestamp when the round was updated.	
answeredInRound	uint80	The round ID in which the answer was computed.	

latestRoundData

solidity

function latestRoundData() external view returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound)

Gets the latest round data.

This function should raise "No data present" if no data is available.

Return Values

Name	Type	Description	
-----	-----		
roundId	uint80	The latest round ID.	
answer	int256	The latest answer.	
startedAt	uint256	The timestamp when the latest round started.	
updatedAt	uint256	The timestamp when the latest round was updated.	
answeredInRound	uint80	The round ID in which the latest answer was computed.	

# index.mdx:

---

section: chainlinkLocal

date: Last Modified

title: "Chainlink Local Data feeds interfaces API reference"

isIndex: true

---

- AggregatorInterface

- AggregatorV2V3Interface

- AggregatorV3Interface

# index.mdx:

---

section: chainlinkLocal

date: Last Modified

title: "Chainlink Local Shared contracts API reference"

isIndex: true

---

- LinkToken

- WETH9

# LinkToken.mdx:

---

section: chainlinkLocal

```
date: Last Modified
title: "LinkToken API reference"
isIndex: true

```

## LinkToken

This contract implements the ChainLink Token (LINK) using the ERC677 standard.

Inherits from the ERC677 token contract and initializes with a fixed total supply and standard token details.

constructor

```
solidity
constructor() public
```

Constructor to initialize the LinkToken contract with a fixed total supply, name, and symbol.

Calls the ERC677 constructor with the name and symbol, and then mints the total supply to the contract deployer.

\onCreate

```
solidity
function onCreate() internal virtual
```

Hook that is called when this contract is created.

Useful to override constructor behaviour in child contracts (e.g., LINK bridge tokens). The default implementation mints 10<sup>27</sup> tokens to the contract deployer.

```
WETH9.mdx:
```

```

section: chainlinkLocal
date: Last Modified
title: "WETH9 API reference"
isIndex: true

```

## WETH9

The WETH9 contract for Wrapped Ether is included in Chainlink Local to help simulate transactions that use native token payments.

name

```
solidity
string name
```

symbol

```
solidity
string symbol
```

decimals

solidity  
uint8 decimals

Approval

solidity  
event Approval(address src, address guy, uint256 wad)

Transfer

solidity  
event Transfer(address src, address dst, uint256 wad)

Deposit

solidity  
event Deposit(address dst, uint256 wad)

Withdrawal

solidity  
event Withdrawal(address src, uint256 wad)

balanceOf

solidity  
mapping(address => uint256) balanceOf

allowance

solidity  
mapping(address => mapping(address => uint256)) allowance

receive

solidity  
receive() external payable

\deposit

solidity  
function deposit() internal

deposit

solidity  
function deposit() external payable

withdraw

solidity  
function withdraw(uint256 wad) external

totalSupply

solidity

function totalSupply() public view returns (uint256)

approve

solidity

function approve(address guy, uint256 wad) public returns (bool)

transfer

solidity

function transfer(address dst, uint256 wad) public returns (bool)

transferFrom

solidity

function transferFrom(address src, address dst, uint256 wad) public returns (bool)

# index.mdx:

---

section: chainlinkLocal

date: Last Modified

title: "Using Chainlink Local to Test CCIP in Your Foundry Project"

isIndex: false

---

import Common from "@features/chainlink-local/Common.astro"

<Common callout="importPackage" />

There are two comprehensive guides to help you test Chainlink CCIP in your Foundry project using Chainlink Local - one for each mode (with and without forking).

We recommend starting with the first guide (without forking) for initial development before advancing to the guide that demonstrates how to test in forked environments.

Guides

Using the CCIP Local Simulator in your Foundry project

This guide helps you set up and run CCIP in a localhost environment within your Foundry project. It provides step-by-step instructions on:

- Cloning the CCIP Foundry Starter Kit
- Installing necessary dependencies
- Running tests for token transfers between two accounts

Using the local simulator without forking is ideal for initial development and testing in an isolated environment.

Using the CCIP Local Simulator in your Foundry project with forked environments

This guide extends your testing capabilities by demonstrating how to use the

CCIP Local Simulator in a forked environment. It includes instructions on:

- Setting up forks of real blockchain networks (such as Arbitrum Sepolia and Ethereum Sepolia)
- Writing test cases
- Running tests for cross-chain token transfers between two accounts

Using the local simulator with forked environments is ideal for testing in a more realistic yet controlled setting.

```
local-simulator-fork.mdx:
```

```

section: chainlinkLocal
date: Last Modified
title: "Using the CCIP Local Simulator in your Foundry project with forked environments"
isIndex: false

```

```
import Common from "@features/chainlink-local/Common.astro"
```

```
<Common callout="importPackage" />
```

You can use Chainlink Local to run CCIP in forked environments within your Foundry project. To get started quickly, you will use the CCIP Foundry Starter Kit. This project includes the `Example01ForkTest` file located in the `./test/fork` directory, demonstrating how to set up and run token transfer tests between two accounts using CCIP in forked environments.

Forked environments allow you to simulate real-world blockchain networks by forking the state of existing chains. In this example, you will fork Arbitrum Sepolia and Ethereum Sepolia.

### Prerequisites

This guide assumes that you are familiar with the guide [Using the CCIP Local Simulator in your Foundry project](#). If not, please get familiar with it and run all the prerequisites.

Set up an `.env` file with the following data:

- `ARBITRUMSEPOLIARPCURL`: The Remote Procedure Call (RPC) URL for the Arbitrum Sepolia network. You can obtain one by creating an account on Alchemy or Infura and setting up an Arbitrum Sepolia project.
- `ETHEREUMSEPOLIARPCURL`: The RPC URL for the Ethereum Sepolia testnet. You can sign up for a personal endpoint from Alchemy, Infura, or another node provider service.

### Test tokens transfers

You will run a test to transfer tokens between two accounts in a forked environment. The test file `Example01ForkTest.t.sol` is located in the `./test/fork` directory.

This file contains two test cases:

1. **Transfer with LINK fees**: This test case transfers tokens from the sender account to the receiver account, paying fees in LINK. At the end of the test, it verifies that the sender account was debited and the receiver account was credited.

1. **Transfer with native gas fees**: This test case transfers tokens from the

sender account to the receiver account, paying fees in native gas. At the end of the test,

it verifies that the sender account was debited and the receiver account was credited.

In these tests, we simulate transfers from a source blockchain (which is a fork of Arbitrum Sepolia) to a destination blockchain (which is a fork of Ethereum Sepolia). Forked environments allow you to simulate real-world blockchain networks by forking the state of existing chains, providing a realistic testing scenario.

For a detailed explanation of the test file, refer to the [Examine the code](#) section.

In your terminal, run the following command to execute the test:

```
shell
forge test --match-contract Example01ForkTest
```

Example output:

```
text
$ forge test --match-contract Example01ForkTest
[â $] Compiling...
No files changed, compilation skipped

Ran 2 tests for test/fork/Example01Fork.t.sol:Example01ForkTest
[PASS] testtransferTokensFromEoaToEoaPayFeesInLink() (gas: 475199)
[PASS] testtransferTokensFromEoaToEoaPayFeesInNative() (gas: 451096)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 11.71s (17.29s
CPU time)

Ran 1 test suite in 11.90s (11.71s CPU time): 2 tests passed, 0 failed, 0
skipped (2 total tests)
```

[Examine the code](#)

Setup

To transfer tokens using CCIP in a forked environment, we need the following:

- Destination chain selector
- Source CCIP router
- LINK token for paying CCIP fees
- A test token contract (such as CCIP-BnM) on both source and destination chains
- A sender account (Alice)
- A receiver account (Bob)

The `setUp()` function is invoked before each test case to reinitialize all the variables, ensuring a clean state for each test:

1. Initialize the source and destination forks:

```
solidity
string memory DESTINATIONRPCURL = vm.envString("ETHEREUMSEPOLIARPCURL");
string memory SOURCERPCURL = vm.envString("ARBITRUMSEPOLIARPCURL");
destinationFork = vm.createSelectFork(DESTINATIONRPCURL);
sourceFork = vm.createFork(SOURCERPCURL);
```

1. Initialize the sender and receiver accounts:

```

solidity
bob = makeAddr("bob");
alice = makeAddr("alice");

```

1. Initialize the fork CCIP local simulator.

Note: `vm.makePersistent` is used to make the `ccipLocalSimulatorFork` address persistent across forks:

```

solidity
ccipLocalSimulatorFork = new CCIPLocalSimulatorFork();
vm.makePersistent(address(ccipLocalSimulatorFork));

```

1. Retrieve and set up the network details for the destination chain.

Note: `Register.NetworkDetails` is a struct that stores network details (such as chain selector, router address, link address, wrapped native address, or CCIP test tokens), and `getNetworkDetails` pulls network details based on chain IDs:

```

solidity
Register.NetworkDetails memory destinationNetworkDetails =
ccipLocalSimulatorFork.getNetworkDetails(block.chainid);
destinationCCIPBnMToken =
BurnMintERC677Helper(destinationNetworkDetails.ccipBnMAddress);
destinationChainSelector = destinationNetworkDetails.chainSelector;

```

1. Switch to the source fork and retrieve the network details for the source chain:

```

solidity
vm.selectFork(sourceFork);
Register.NetworkDetails memory sourceNetworkDetails =
ccipLocalSimulatorFork.getNetworkDetails(block.chainid);
sourceCCIPBnMToken =
BurnMintERC677Helper(sourceNetworkDetails.ccipBnMAddress);
sourceLinkToken = IERC20(sourceNetworkDetails.linkAddress);
sourceRouter = IRouterClient(sourceNetworkDetails.routerAddress);

```

1. All the variables are stored in the contract state for use in the test cases.

Prepare scenario (helper function)

The `prepareScenario()` function is invoked at the beginning of each test case. It performs the following actions:

1. Select the source fork and request CCIP-BnM tokens for Alice:

```

solidity
vm.selectFork(sourceFork);
vm.startPrank(alice);
sourceCCIPBnMToken.drip(alice);

```

1. Approve the source router to spend tokens on behalf of Alice:

```

solidity
uint256 amountToSend = 100;
sourceCCIPBnMToken.approve(address(sourceRouter), amountToSend);

```

1. Create an array `Client.EVMTokenAmount[]` to specify the token transfer details. This array and the amount to send are returned by the `prepareScenario()` function for use in the calling test case:

```
solidity
tokensToSendDetails = new Client.EVMTokenAmount[](1);
Client.EVMTokenAmount memory tokenToSendDetails =
 Client.EVMTokenAmount({token: address(ccipBnMToken), amount:
amountToSend});
tokensToSendDetails[0] = tokenToSendDetails;
```

1. Stop impersonating Alice (sender):

```
solidity
vm.stopPrank();
```

Test case 1: Transfer with LINK fees

The `testtransferTokensFromEoaToEoaPayFeesInLink` function tests the transfer of tokens between two externally owned accounts (EOA) while paying fees in LINK. Here are the steps involved in this test case:

1. Invoke the `prepareScenario()` function to set up the necessary variables:

```
solidity
(Client.EVMTokenAmount[] memory tokensToSendDetails, uint256 amountToSend) =
prepareScenario();
```

1. Select the destination fork and record the initial token balance of Bob receiver:

```
solidity
vm.selectFork(destinationFork);
uint256 balanceOfBobBefore = destinationCCIPBnMToken.balanceOf(bob);
```

1. Select the source fork and record the initial token balance of Alice (sender):

```
solidity
vm.selectFork(sourceFork);
uint256 balanceOfAliceBefore = sourceCCIPBnMToken.balanceOf(alice);
```

1. Request 10 LINK tokens from the CCIP local simulator faucet for Alice (sender):

```
solidity
ccipLocalSimulatorFork.requestLinkFromFaucet(alice, 10 ether);
```

1. Construct the `Client.EVM2AnyMessage` structure with the receiver, token amounts, and other necessary details.

- Set the data parameter to an empty string because you are not sending any arbitrary data, only tokens.
- In `extraArgs`, set the gas limit to 0. This gas limit is for execution of receiver logic, which doesn't apply here because you're sending tokens to an EOA.



```

solidity
Client.EVM2AnyMessage memory message = Client.EVM2AnyMessage({
 receiver: abi.encode(bob),
 data: abi.encode(""),
 tokenAmounts: tokensToSendDetails,
 extraArgs: Client.argsToBytes(Client.EVMExtraArgsV1({gasLimit: 0})),
 feeToken: address(sourceLinkToken)
});

```

1. Calculate the required fees for the transfer and approve the router to spend LINK tokens for these fees:

```

solidity
uint256 fees = sourceRouter.getFee(destinationChainSelector, message);
sourceLinkToken.approve(address(sourceRouter), fees);

```

1. Send the CCIP transfer request to the router:

```

solidity
sourceRouter.ccipSend(destinationChainSelector, message);

```

1. Stop impersonating Alice (sender):

```

solidity
vm.stopPrank();

```

1. Record Alice's final token balance and verify that it has decreased by the amount sent:

```

solidity
uint256 balanceOfAliceAfter = sourceCCIPBnMToken.balanceOf(alice);
assertEq(balanceOfAliceAfter, balanceOfAliceBefore - amountToSend);

```

1. Call the switchChainAndRouteMessage function to switch to the destination fork and route the message to complete the transfer:

```

solidity
ccipLocalSimulatorFork.switchChainAndRouteMessage(destinationFork);

```

1. Record Bob's final token balance and verify that it has increased by the amount sent:

```

solidity
uint256 balanceOfBobAfter = destinationCCIPBnMToken.balanceOf(bob);
assertEq(balanceOfBobAfter, balanceOfBobBefore + amountToSend);

```

## Test case 2: Transfer with native gas fees

The `testtransferTokensFromEoaToEoaPayFeesInNative` function tests the transfer of tokens between two externally owned accounts (EOA) while paying fees in native gas.

Here are the steps involved in this test case:

1. Invoke the `prepareScenario()` function to set up the necessary variables. (This function is the same as in the previous test case.)
1. Select the destination fork and record Bob's initial token balance. (This

step is the same as in the previous test case.)

1. Select the source fork and record Alice's initial token balance. (This step is the same as in the previous test case.)

1. Begin impersonating Alice (sender) and provide her with native gas:

```
solidity
vm.startPrank(alice);
deal(alice, 5 ether);
```

1. Construct the Client.EVM2AnyMessage structure. This step is the same as in the previous test case.

The main difference is that the feeToken is set with address(0) to indicate that the fees are paid in native gas:

```
solidity
Client.EVM2AnyMessage memory message = Client.EVM2AnyMessage({
 receiver: abi.encode(bob),
 data: abi.encode(""),
 tokenAmounts: tokensToSendDetails,
 extraArgs: Client.argsToBytes(Client.EVMExtraArgsV1({gasLimit: 0})),
 feeToken: address(0)
});
```

1. Calculate the required fees for the transfer and send the CCIP transfer request along with the necessary native gas:

```
solidity
uint256 fees = sourceRouter.getFee(destinationChainSelector, message);
sourceRouter.ccipSend{value: fees}(destinationChainSelector, message, fees);
```

1. Stop impersonating Alice (sender). (This step is the same as in the previous test case.)

1. Verify Alice's (sender) balance. (This step is the same as in the previous test case.)

1. Call the switchChainAndRouteMessage function

to switch to the destination fork and route the message to complete the transfer. (This step is the same as in the previous test case.)

1. Verify Bob's (receiver) balance. (This step is the same as in the previous test case.)

# local-simulator.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Using CCIP local simulator in your Foundry project"
isIndex: false

```

```
import Common from "@features/chainlink-local/Common.astro"
```

```
<Common callout="importPackage" />
```

You can use Chainlink Local to run CCIP in a localhost environment within your Foundry project. To get started quickly, you will use the CCIP Foundry Starter Kit. This project is a Foundry boilerplate that includes the Chainlink Local package and several CCIP examples.

Prerequisites

1. In a terminal, clone the CCIP Foundry Starter Kit repository and change directories.

```
shell
git clone https://github.com/smartcontractkit/ccip-starter-kit-foundry.git &&
\
cd ./ccip-starter-kit-foundry/
```

1. Run `npm install` to install the dependencies. This command installs the Chainlink Local package and other required packages:

```
shell
npm install
```

1. Run `forge install` to install packages:

```
shell
forge install
```

1. Run `forge build` to compile the contracts:

```
shell
forge build
```

#### Test tokens transfers

You will run a test to transfer tokens between two accounts. The test file `Example01.t.sol` is located in the `./test/no-fork` directory. This file contains two test cases:

1. Transfer with LINK fees: This test case transfers tokens from the sender account to the receiver account, paying fees in LINK. At the end of the test, it verifies that the sender account was debited and the receiver account was credited.

1. Transfer with native gas fees: This test case transfers tokens from the sender account to the receiver account, paying fees in native gas. At the end of the test, it verifies that the sender account was debited and the receiver account was credited.

For a detailed explanation of the test file, refer to the [Examine the code](#) section.

In your terminal, run the following command to execute the test:

```
shell
forge test --match-contract Example01Test
```

Example output:

```
text
$ forge test --match-contract Example01Test
[â $] Compiling...
No files changed, compilation skipped

Ran 2 tests for test/no-fork/Example01.t.sol:Example01Test
[PASS] testtransferTokensFromEoaToEoaPayFeesInLink() (gas: 167576)
```

```
[PASS] testtransferTokensFromEoaToEoaPayFeesInNative() (gas: 122348)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 8.79ms
(976.54µs CPU time)
```

Ran 1 test suite in 201.00ms (8.79ms CPU time): 2 tests passed, 0 failed, 0 skipped (2 total tests)

Examine the code

Setup

To transfer tokens using CCIP, we need the following:

- Destination chain selector
- Source CCIP router
- LINK token for paying CCIP fees
- A test token contract (such as CCIP-BnM)
- A sender account (Alice)
- A receiver account (Bob)

The `setUp()` function is invoked before each test case to reinitialize all the variables, ensuring a clean state for each test:

1. Initialize the CCIP local simulator contract:

```
solidity
ccipLocalSimulator = new CCIPLocalSimulator();
```

1. Invoke the configuration function to retrieve the configuration details for the pre-deployed contracts and services needed for local CCIP simulations:

```
solidity
(
 uint64 chainSelector,
 IRouterClient sourceRouter,
 ,
 ,
 LinkToken linkToken,
 BurnMintERC677Helper ccipBnM,
) = ccipLocalSimulator.configuration();
```

Note: The configuration function also returns the destination router, WETH9, and CCIP-LnM contracts, but we are not using them in these test cases. Hence, there are commas in the return values.

1. Initialize the sender and receiver accounts:

```
solidity
alice = makeAddr("alice")
Bob = makeAddr("Bob")
```

1. All the variables are stored in the contract state for use in the test cases.

Prepare scenario (helper function)

The `prepareScenario()` function is invoked at the beginning of each test case. It performs the following actions:

1. Request CCIP-BnM tokens for Alice (sender):

```
solidity
ccipBnMToken.drip(alice);
```

1. Approve the source router to spend tokens on behalf of Alice (sender):

```
solidity
amountToSend = 100;
ccipBnMToken.approve(address(router), amountToSend);
```

1. Create an array `Client.EVMTokenAmount[]` to specify the token transfer details. This array and the amount to send are returned by the `prepareScenario()` function for use in the calling test case:

```
solidity
tokensToSendDetails = new Client.EVMTokenAmount[](1);
Client.EVMTokenAmount memory tokenToSendDetails =
 Client.EVMTokenAmount({token: address(ccipBnMToken), amount:
amountToSend});
tokensToSendDetails[0] = tokenToSendDetails;
```

#### Test case 1: Transfer with LINK fees

The `testtransferTokensFromEoaToEoaPayFeesInLink` function tests the transfer of tokens between two externally owned accounts (EOA) while paying fees in LINK. Here are the steps involved in this test case:

1. Invoke the `prepareScenario()` function to set up the necessary variables:

```
solidity
(Client.EVMTokenAmount[] memory tokensToSendDetails, uint256 amountToSend) =
prepareScenario();
```

1. Record the initial token balances for Alice (sender) and Bob (receiver):

```
solidity
uint256 balanceOfAliceBefore = ccipBnMToken.balanceOf(alice);
uint256 balanceOfBobBefore = ccipBnMToken.balanceOf(bob);
```

1. Begin impersonating Alice (sender) to perform the subsequent actions:

```
solidity
vm.startPrank(alice);
```

1. Request 5 LINK tokens from the CCIP Local Simulator faucet for Alice (sender):

```
solidity
ccipLocalSimulator.requestLinkFromFaucet(alice, 5 ether);
```

1. Construct the `Client.EVM2AnyMessage` structure with the receiver, token amounts, and other necessary details.

- Set the data parameter to an empty string because you are not sending any arbitrary data, only tokens.
- In `extraArgs`, set the gas limit to 0. This gas limit is for execution of receiver logic, which doesn't apply here because you're sending tokens to an

EOA.

```
solidity
Client.EVM2AnyMessage memory message = Client.EVM2AnyMessage({
 receiver: abi.encode(bob),
 data: abi.encode(""),
 tokenAmounts: tokensToSendDetails,
 extraArgs: Client.argsToBytes(Client.EVMExtraArgsV1({gasLimit: 0})),
 feeToken: address(linkToken)
});
```

1. Calculate the required fees for the transfer and approve the router to spend LINK tokens for these fees:

```
solidity
uint256 fees = router.getFee(destinationChainSelector, message);
linkToken.approve(address(router), fees);
```

1. Send the CCIP transfer request to the router:

```
solidity
router.ccipSend(destinationChainSelector, message);
```

1. Stop impersonating (sender):

```
solidity
vm.stopPrank();
```

1. Record the final token balances for Alice (sender) and Bob (receiver):

```
solidity
uint256 balanceOfAliceAfter = ccipBnMToken.balanceOf(alice);
uint256 balanceOfBobAfter = ccipBnMToken.balanceOf(bob);
```

1. Verify that Alice's balance has decreased by the amount sent and Bob's balance has increased by the same amount:

```
solidity
assertEq(balanceOfAliceAfter, balanceOfAliceBefore - amountToSend);
assertEq(balanceOfBobAfter, balanceOfBobBefore + amountToSend);
```

## Test case 2: Transfer with native gas fees

The `testtransferTokensFromEoaToEoaPayFeesInNative` function tests the transfer of tokens between two externally owned accounts (EOA) while paying fees in native gas.

Here are the steps involved in this test case:

1. Invoke the `prepareScenario()` function to set up the necessary variables. (This step is the same as in the previous test case.)
1. Record the initial token balances of Alice (sender) and Bob (receiver). (This step is the same as in the previous test case.)
1. Begin impersonating Alice (sender) and provide her with native gas to pay for the fees:

```
solidity
vm.startPrank(alice);
deal(alice, 5 ether);
```

1. Construct the `Client.EVM2AnyMessage` structure. This step is the same as in the previous test case.

The main difference is that the `feeToken` is set with `address(0)` to indicate that the fees are paid in native gas:

```
solidity
Client.EVM2AnyMessage memory message = Client.EVM2AnyMessage({
 receiver: abi.encode(bob),
 data: abi.encode(""),
 tokenAmounts: tokensToSendDetails,
 extraArgs: Client.argsToBytes(Client.EVMExtraArgsV1({gasLimit: 0})),
 feeToken: address(0)
});
```

1. Calculate the required fees for the transfer and send the CCIP transfer request along with the necessary native gas:

```
solidity
uint256 fees = router.getFee(destinationChainSelector, message);
router.ccipSend{value: fees}(destinationChainSelector, message);
```

1. Stop impersonating Alice (sender) and verify the token balances for Alice (sender) and Bob (receiver). (This step is the same as in the previous test case.)

Next steps

For more advanced scenarios, please refer to other test files in the `./test/no-fork` directory. To learn how to use Chainlink Local in forked environments, refer to the guide on [Using the CCIP Local Simulator in your Foundry project with forked environments](#).

# index.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Using Chainlink Local to Test CCIP in Your Hardhat Project"
isIndex: false

```

```
import Common from "@features/chainlink-local/Common.astro"
```

```
<Common callout="importPackage" />
```

There are two comprehensive guides to help you test Chainlink CCIP in your Hardhat project using Chainlink Local - one for each mode (with and without forking).

We recommend starting with the first guide (without forking) for initial development before advancing to the guide that demonstrates how to test in forked environments.

Guides

Using CCIP Local Simulator in your Hardhat project

This guide helps you set up and run CCIP in a localhost environment within your Hardhat project. It provides step-by-step instructions on :

- Cloning the CCIP Hardhat Starter Kit
- Installing necessary dependencies
- Running tests for token transfers between two accounts

Using the local simulator without forking is ideal for initial development and testing in an isolated environment.

Using CCIP Local Simulator in your Hardhat project with forked environments

This guide extends your testing capabilities by demonstrating how to use the CCIP Local Simulator in a forked environment. It includes instructions on:

- Setting up forks of real blockchain networks (e.g., Arbitrum Sepolia and Ethereum Sepolia)
- Writing test cases
- Running tests for cross-chain token transfers between two accounts

Using the local simulator with forked environments is ideal for testing in a more realistic yet controlled setting.

```
local-simulator-fork.mdx:
```

```

section: chainlinkLocal
date: Last Modified
title: "Using CCIP Local Simulator in Your Hardhat Project - Forked
Environments"
isIndex: false

```

```
import Common from "@features/chainlink-local/Common.astro"
```

```
<Common callout="importPackage" />
```

You can use Chainlink Local to run CCIP in forked environments within your Hardhat project. To get started quickly, you will use the CCIP Hardhat Starter Kit.

This project includes the Example1.spec.tst file located in the ./test/fork directory, demonstrating how to set up and run token transfer tests between two accounts using CCIP in forked environments.

Forked environments allow you to simulate real-world blockchain networks by forking the state of existing chains. In this example, we will fork Arbitrum Sepolia and Ethereum Sepolia.

### Prerequisites

This guide assumes that you are familiar with the guide Using CCIP Local Simulator in Your Hardhat Project.

If not, please get familiar with it and run all the prerequisites.

Set an environment variable file. For higher security, the examples repository imports @chainlink/env-enc.

Use this tool to encrypt your environment variables at rest.

1. Set an encryption password for your environment variables.

```
shell
npx env-enc set-pw
```



1. Run `npx env-enc set` to configure a `.env.enc` file with:

- `ARBITRUMSEPOLIARPCURL`: The Remote Procedure Call (RPC) URL for the Arbitrum Sepolia network. You can obtain one by creating an account on Alchemy or Infura and setting up an Arbitrum Sepolia project.
- `ETHEREUMSEPOLIARPCURL`: The RPC URL for the Ethereum Sepolia testnet. You can sign up for a personal endpoint from Alchemy, Infura, or another node provider service.

Test tokens transfers

You will run a test to transfer tokens between two accounts in a forked environment. The test file `Example1.spec.ts` is located in the `./test/fork` directory.

This file contains one test case:

**Transfer with LINK fees:** This test case transfers tokens from the sender account to the receiver account, paying fees in LINK. At the end of the test, it verifies that the sender account was debited and the receiver account was credited.

In this test, we simulate a transfer of tokens from an Externally Owned Account (EOA) on a source blockchain (which is a fork of Arbitrum Sepolia) to an EOA on a destination blockchain (which is a fork of Ethereum Sepolia). Forked environments allow you to simulate real-world blockchain networks by forking the state of existing chains, providing a realistic testing scenario.

For a detailed explanation of the test file, refer to the [Examine the code](#) section.

In your terminal, run the following command to execute the test:

```
shell
npx hardhat test test/fork/Example1.spec.ts
```

Example output:

```
text
$ npx hardhat test test/fork/Example1.spec.ts
```

```
Example 1 - Fork
 â€œ Should transfer CCIP test tokens from EOA to EOA (10889ms)
```

```
1 passing (11s)
```

[Examine the code](#)

To transfer tokens using CCIP in a forked environment, we need the following:

- Destination chain selector
- Source CCIP router
- LINK token for paying CCIP fees
- A test token contract (such as CCIP-BnM) on both source and destination chains
- A sender account (Alice)
- A receiver account (Bob)

The `it("Should transfer CCIP test tokens from EOA to EOA")` function sets up the necessary environment and runs the test. Here are the steps involved:

1. Initialize the sender and receiver accounts:

```
typescript
const [alice, bob] = await hre.ethers.getSigners()
```

1. Set the source and destination chains:

```
typescript
const [source, destination] = ["ethereumSepolia", "arbitrumSepolia"]
```

1. Retrieve the necessary addresses and configurations (such as the LINK token address, source router address, and so on).

1. Fork the source network:

```
typescript
await hre.network.provider.request({
 method: "hardhatreset",
 params: [
 {
 forking: {
 jsonRpcUrl: getProviderRpcUrl(source),
 },
 },
],
})
```

1. Connect to the source router and CCIP-BnM contracts.

1. Call sourceCCIPBnM.drip to request CCIP-BnM tokens for Alice (sender).

1. Approve the source router to spend tokens on behalf of Alice (sender).

1. Construct the Client.EVM2AnyMessage structure. This step is similar to the non fork example.

1. Call the source router to estimate the fees.

1. Call the requestLinkFromTheFaucet function to request LINK tokens for Alice (sender).

1. Connect to the LINK contract and approve the LINK token for paying CCIP fees.

1. Estimate Alice (sender) balance before the transfer.

1. Call the source router to send the CCIP request.

1. Wait for the transaction to be included in a block.

1. Call the getEvm2EvmMessage function to parse the transaction receipt and extract the CCIPSendRequested event and then decodes it to an object.

1. Verify that Alice's balance has decreased by the amount sent.

1. Fork and switch to the destination network:

```
typescript
await hre.network.provider.request({
 method: "hardhatreset",
```

```

 params: [
 {
 forking: {
 jsonRpcUrl: getProviderRpcUrl(destination),
 },
 },
],
 },
}
})

```

1. Call the `routeMessage` function to route the message to the destination router.
1. Connect to the CCIP-BnM contract using the CCIP-BnM destination address.
1. Verify that Bob's balance has increased by the amount sent.

# local-simulator.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Using CCIP local simulator in your Hardhat project"
isIndex: false

```

import Common from "@features/chainlink-local/Common.astro"

<Common callout="importPackage" />

You can use Chainlink Local to run CCIP in a localhost environment within your Hardhat project. To get started quickly, you will use the CCIP Hardhat Starter Kit. This project is a Hardhat boilerplate that includes the Chainlink Local package and several CCIP examples.

#### Prerequisites

1. In a terminal, clone the CCIP Hardhat Starter Kit repository and change directories:

```

shell
git clone https://github.com/smartcontractkit/ccip-starter-kit-hardhat && \
cd ./ccip-starter-kit-hardhat/

```

1. Install the Chainlink Local package and other required packages:

```

shell
npm install

```

1. Compile the contracts:

```

shell
npm run compile

```

#### Test tokens transfers

You will run a test to transfer tokens between two accounts. The test file `Example1.spec.ts` is located in the `./test/no-fork` directory. This file contains one test case:

Transfer with LINK fees: This test case transfers tokens from the sender account to the receiver account, paying fees in LINK. At the end of the test, it verifies that the sender account was debited and the receiver account was credited.

For a detailed explanation of the test file, refer to the Examine the code section.

In your terminal, run the following command to execute the test:

```
shell
npx hardhat test test/no-fork/Example1.spec.ts
```

Example output:

```
text
$ npx hardhat test test/no-fork/Example1.spec.ts
```

```
Example 1
 "Should transfer CCIP test tokens from EOA to EOA (1057ms)"
```

```
1 passing (1s)
```

Examine the code

Setup

To transfer tokens using CCIP, we need the following:

- Destination chain selector
- Source CCIP router
- LINK token for paying CCIP fees
- A test token contract (such as CCIP-BnM)
- A sender account (Alice)
- A receiver account (Bob)

The `deployFixture` function is used to set up the initial state for the tests. This function deploys the CCIP local simulator contract and initializes the sender and receiver accounts.

1. Initialize the CCIP local simulator contract:

```
typescript
const ccipLocalSimualtorFactory = await
hre.ethers.getContractFactory("CCIPLocalSimulator")
const ccipLocalSimulator: CCIPLocalSimulator = await
ccipLocalSimualtorFactory.deploy()
```

1. Initialize the sender and receiver accounts:

```
typescript
const [alice, bob] = await hre.ethers.getSigners()
```

Test case: Transfer with LINK fees

The `it("Should transfer CCIP test tokens from EOA to EOA")` function tests the transfer of tokens between two externally owned accounts (EOA)

while paying fees in LINK. Here are the steps involved in this test case:

1. Invoke the `deployFixture` function to set up the necessary variables:

```
typescript
const { ccipLocalSimulator, alice, bob } = await loadFixture(deployFixture)
```

1. Invoke the configuration function to retrieve the configuration details for the pre-deployed contracts and services needed for local CCIP simulations.

1. Connect to the source router and CCIP-BnM contracts.

1. Call `ccipBnM.drip` to request CCIP-BnM tokens for Alice (sender).

1. Create an array `Client.EVMTokenAmount[]` to specify the token transfer details:

```
typescript
const tokenAmounts = [
 {
 token: config.ccipBnM,
 amount: amountToSend,
 },
]
```

1. Construct the `Client.EVM2AnyMessage` structure with the receiver, token amounts, and other necessary details.

- Use an empty string for the data parameter because you are not sending any arbitrary data (only tokens).
- Set `gasLimit` to 0 because you are sending tokens to an EOA, which means you do not expect any execution of receiver logic (and therefore do not need gas for that).

```
typescript
const gasLimit = 0
const functionSelector = id("CCIP EVMExtraArgsV1").slice(0, 10)
const defaultAbiCoder = AbiCoder.defaultAbiCoder()
const extraArgs = defaultAbiCoder.encode(["uint256"], [gasLimit])
const encodedExtraArgs = `${functionSelector}${extraArgs.slice(2)}`

const message = {
 receiver: defaultAbiCoder.encode(["address"], [bob.address]),
 data: defaultAbiCoder.encode(["string"], [""]), // no data
 tokenAmounts: tokenAmounts,
 feeToken: config.linkToken,
 extraArgs: encodedExtraArgs,
}
```

1. Calculate the required fees for the transfer and approve the router to spend LINK tokens for these fees:

```
typescript
const fee = await mockCcipRouter.getFee(config.chainSelector, message)
await linkToken.connect(alice).approve(mockCcipRouterAddress, fee)
```

1. Send the CCIP transfer request to the router:

```
typescript
await mockCcipRouter.connect(alice).ccipSend(config.chainSelector, message)
```

1. Verify that Alice's balance has decreased by the amount sent and Bob's balance has increased by the same amount:

```
typescript
expect(await ccipBnM.balanceOf(alice.address)).to.deep.equal(ONEETHER -
amountToSend)
expect(await ccipBnM.balanceOf(bob.address)).to.deep.equal(amountToSend)
```

Next steps

For more advanced scenarios, please refer to other test files in the `./test/no-fork` directory. To learn how to use Chainlink local in forked environments, refer to the guide on Using CCIP Local Simulator in your Hardhat project with forked environments.

# index.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Using Chainlink Local to Test CCIP in RemixIDE"
isIndex: false

```

```
import Common from "@features/chainlink-local/Common.astro"
```

```
<Common callout="importPackage" />
```

There is one comprehensive guide to help you test Chainlink CCIP in Remix IDE using Chainlink Local - without forking. Testing in forked environments is currently supported only for Foundry and Hardhat.

Guides

Using CCIP Local Simulator in Remix IDE

This guide helps you to test Chainlink CCIP getting started guide locally in Remix IDE. You will:

- Deploy a CCIP sender contract
- Deploy CCIP receiver contract
- Use the local simulator to send data from the sender contract to the receiver contract

# local-simulator.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Using the CCIP local simulator in Remix IDE"
isIndex: false

```

```
import Common from "@features/chainlink-local/Common.astro"
import { Aside, CodeSample, ClickToZoom, CopyText } from "@components"
```

```
<Common callout="importPackage" />
```

In this guide, you will test the Chainlink CCIP getting started guide locally in

Remix IDE. You will:

- Deploy a CCIP sender contract
- Deploy a CCIP receiver contract
- Use the local simulator to send data from the sender contract to the receiver contract

### Prerequisites

Remix IDE is an online development environment that allows you to write, deploy, and test smart contracts. By default Remix IDE does not persist the files that you open from an external source. To save files, you will need to manually create a workspace and copy the files into the workspace.

1. Open the Remix IDE in your browser.
1. Create a new workspace.
1. Copy the content of the Test CCIP Local Simulator contract into a new file in the workspace.

```
<CodeSample src="samples/CCIP/TestCCIPLocalSimulator.sol" />
```

1. Copy the content of the Sender contract into a new file in the workspace.

```
<CodeSample src="samples/CCIP/Sender.sol" />
```

1. Copy the content of the Receiver contract into a new file in the workspace.

```
<CodeSample src="samples/CCIP/Receiver.sol" />
```

At this point, you should have three files in your workspace:

- TestCCIPLocalSimulator.sol: The file imports the Chainlink CCIP local simulator contract.
- Sender.sol: The file contains the Sender contract that interacts with CCIP to send data to the Receiver contract.
- Receiver.sol: The file contains the Receiver contract that receives data from the Sender contract.

### Deploy the contracts

1. Compile the contracts.
1. Under the Deploy & Run Transactions tab, make sure Remix VM is selected in the Environment drop-down list. Remix will use a sandbox blockchain in the browser to deploy the contracts.
1. Deploy the CCIP local simulator:

1. Select the TestCCIPLocalSimulator.sol file in the file explorer.
1. In the Contract drop-down list, select CCIPLocalSimulator.
1. Click the Deploy button.
1. The CCIPLocalSimulator is shown in the Deployed Contracts section.
1. In the list of functions, click the configuration function to retrieve the

configuration details for the pre-deployed contracts and services needed for local CCIP simulations:

```
<ClickToZoom
 src="/images/chainlink-local/ccip/remix/cciplocalsimulator-
configuration.jpg"
 alt="Remix IDE CCIP Local Simulator configuration"
/>
```

1. You will interact with the LINK token contract to fund the sender contract with LINK tokens. The LINK token contract is pre-deployed in the local simulator configuration, so you can simply load the LINK token contract instance:

1. Select LinkToken in the Contract drop-down list.
1. Fill in the At Address field with the address of the LINK token contract from the CCIPLocalSimulator configuration.
1. Click the At Address button.

The LinkToken contract is shown in the Deployed Contracts section.

1. Deploy the Sender.sol contract:

1. Select the Sender.sol file in the file explorer.
1. In the Contract drop-down list, select Sender.
1. Under the Deploy section, fill in the constructor parameters:
  - router: The address of the sourceRouter contract from the CCIPLocalSimulator configuration.
  - link: The address of the LINK token contract from the CCIPLocalSimulator configuration.
1. Click the Deploy button.

The Sender contract is shown in the Deployed Contracts section.

1. Deploy the Receiver.sol contract:

1. Select the Receiver.sol file in the file explorer.
1. In the Contract drop-down list, select Receiver.
1. Under the Deploy section, fill in the constructor parameters:
  - router: The address of the destinationRouter contract from the CCIPLocalSimulator configuration.
1. Click the Deploy button.

The Receiver contract is shown in the Deployed Contracts section.

Transfer data from the sender to the receiver

1. Fund the sender contract with LINK tokens to pay for CCIP fees:

1. Copy the address of the Sender contract from the Deployed Contracts section.
1. In the CCIPLocalSimulator contract, fill in the requestLINKFromFaucet function with the following inputs:
  - to: The address of the Sender contract.
  - amount: The amount of LINK tokens to transfer. For instance: `<CopyText text="1000000000000000000" code/>`.
1. Click the Transact button.

1. Send data from the sender contract to the receiver contract:

1. Copy the address of the Receiver contract from the Deployed Contracts section.
1. In the Sender contract, fill in the sendMessage function with:
  - destinationChainSelector: The destination chain selector. You can find it in the CCIPLocalSimulator configuration.
  - receiver: The address of the Receiver contract.
  - text: The text to send. For instance `<CopyText text="Hello!" code/>`.
1. Remix IDE fails to estimate the gas properly for the sendMessage function. To work around this, you need to set the gas limit manually to `<CopyText text="3000000" code/>`:

`<ClickToZoom`



```
src="/images/chainlink-local/ccip/remix/sendMessage-estimateGas.jpg"
alt="RemixIDE CCIP Local sendMessage force gas limit"
/>
```

1. Click the Transact button.

1. Check the receiver contract to verify the data transfer:

1. In the Receiver contract, click on the `getLastReceivedMessageDetails` function.

1. The `getLastReceivedMessageDetails` function returns the text sent from the Sender contract:

```
<ClickToZoom
src="/images/chainlink-local/ccip/remix/receivedMessage.jpg"
alt="RemixIDE CCIP Local receivedMessage"
/>
```

Next steps

You have successfully tested the CCIP getting started guide within a few minutes using Remix IDE. Testing locally is useful to debug your contracts and fix any issues before testing them on testnets, saving you time and resources. As an exercise, you can try any of the CCIP guides using the local simulator in Remix IDE.

# architecture.mdx:

```

section: chainlinkLocal
date: Last Modified
title: "Chainlink Local Architecture"
isIndex: false

```

```
import { ClickToZoom, Aside } from "@components"
```

Chainlink Local is a package that you can import into your development environment (such as Hardhat, Foundry, or Remix IDE) to use Chainlink services locally. It supports two primary modes:

- Local testing without forking.
- Local testing with forking.

This enables you to test Chainlink smart contracts and services, such as CCIP, either in a clean local blockchain state or by using a forked state from a live blockchain.

After testing with Chainlink Local, you can deploy your contracts to test networks without any modifications.

Local testing without forking

In this mode, you work with mock contracts on a locally running development blockchain node, such as Hardhat, Anvil (Foundry), or Remix VM.

How it works: Import the Chainlink Local package and deploy a set of Chainlink smart contracts to a blank Hardhat/Anvil network EVM state as part of your tests.

You can then deploy your smart contracts and start testing with them.

```
<ClickToZoom
```

```
src="/images/chainlink-local/chainlink-local-no-fork.png"
alt="Chainlink Local Architecture (Without Forking)"
/>
```

## Local testing with forking

In this mode, you work with deployed Chainlink smart contracts using one or more forked blockchains. This setup provides a more realistic testing environment by incorporating the state of a live blockchain.

<Aside type="note" title="Supported Development environments">

Chainlink Local does not support forking with Remix IDE. Forking is currently supported only for Hardhat and Foundry.

</Aside>

How it works: In your test scripts, you fork one or more blockchains to create locally running blockchain network(s). Chainlink Local provides the necessary interfaces

to interact with the forked Chainlink services and utilities for testing in a forked environment. For example, in CCIP, Chainlink Local provides a utility function that helps

you switch from one fork (as the source chain) to another fork (as the destination chain).

```
<ClickToZoom
src="/images/chainlink-local/chainlink-local-fork.png"
alt="Chainlink Local Architecture (With Forking)"
/>
```

# contributing.mdx:

---

section: chainlinkLocal

date: Last Modified

title: "Contributing to Chainlink Local"

isIndex: true

---

Contributions to Chainlink Local are welcome! These contributions can include bug fixes, new features, and documentation updates. This guide provides a list of repositories you can contribute to:

- Chainlink Local: The main repository for Chainlink Local.
- CCIP Foundry Starter Kit: A Foundry starter kit for Chainlink CCIP. It uses Chainlink Local test different scenarios on a local environment with and without forking.
- CCIP Hardhat Starter Kit: A Hardhat starter kit for Chainlink CCIP. It uses Chainlink Local test different scenarios on a local environment with and without forking.
- Chainlink Technical Documentation: You are welcome to write new guides or update existing guides in the Chainlink documentation.

# configuring-nodes.mdx:

---

section: nodeOperator

date: Last Modified

title: "Configuring Nodes"

---

Starting with Chainlink v2.0.0, TOML configuration is the only supported

configuration method. You can switch to TOML format using the Chainlink v1.13.0 image. After your Chainlink node is running stable on v1.13.0, you can continue to use the TOML config on future images where support for .env configs are no longer supported.

You can see the available config options on the [Node Config](#) and [Node Secrets](#) pages.

### Migrating from environment variables to TOML

Before you begin, update your Chainlink Node to v1.13.0 and ensure that it operates as expected. This is the last version that supports environment variable configuration, and the best version to use for TOML migration before you update the node to future versions.

#### Export your current config

You can generate the config.toml file using the chainlink config dump command. This guide shows you how to do this for a Chainlink node running in a Docker container:

1. Open an interactive shell inside the Docker container that runs your node:

```
shell
docker exec -it chainlink bash
```

1. Log in to the Chainlink CLI with an account that has admin access:

```
shell
chainlink admin login
```

1. Export the current config to a config.toml file:

```
shell
chainlink config dump > config.toml
```

1. Log out of the Chainlink CLI and close the shell on the Docker container.

```
shell
chainlink admin logout && exit
```

1. Change to the directory where your .env file is located. This is the directory that you mount to Docker when you run the node. You will create your config.toml and secrets.toml files here so Docker can provide them to your node container.

```
shell
cd /.chainlink
```

1. Write the config file from the container to your host:

```
shell
docker exec -it chainlink cat /home/chainlink/config.toml > ./config.toml
```

1. Create a secrets.toml file with the minimum required secrets, which are [Database] and [Password].

If you are working on a test node, you can use ?sslmode=disable in the

database URL. You might also need `AllowSimplePasswords = true` in the `[Database]` section so you can start the node, but you should make the database password sufficiently complex as a best practice:

```
shell
echo "[Database]
URL = 'postgresql://user:pass@localhost:5432/dbname'

[Password]
Keystore = 'keystorepass'" > ./secrets.toml
```

1. Edit the `secrets.toml` file to include the secrets you need for your specific Chainlink node. See the [Node Secrets](#) page for a full list of available options.

#### Validate the configuration

After you have the `config.toml` file and the `secrets.toml` files on the host system, you can validate these files using a temporary Docker container.

1. Validate the configuration by running the `config validate` command in the Docker container. This command changes to `node validate` when you upgrade your node to version 2.0.0 or later.

```
shell
docker run --platform linux/x8664/v8 --name chainlink-config-validator -
v /.chainlink:/chainlink -it --rm smartcontract/chainlink:1.13.0 -config
/chainlink/config.toml -secrets /chainlink/secrets.toml config validate
```

You will likely see some invalid config errors. For example:

```
shell
Invalid configuration: EVM.3.Nodes: missing: must have at least one node
```

1. Edit the `config.toml` and `secrets.toml` files to manually correct any errors. See the [Node Config](#) and [Node Secrets](#) pages to learn which settings are valid. For the error in this example, an EVM chain was configured with no nodes. Removing this from the config made the config valid:

```
[[EVM]]
ChainID = '421613'
Enabled = false
Nodes = []
```

1. Run the `config validate` command again and make additional changes until you have a valid config message:

```
shell
Valid configuration.
```

#### Restart your Chainlink node using the TOML config

With your valid config and secrets files, you can migrate your Chainlink node to use the new config.

1. Stop your existing Chainlink node:

```
shell
docker stop chainlink
```

1. Make a Postgres database snapshot so you can restore your previous Chainlink node if necessary.

1. Start a new Chainlink node Docker container named using the new config.toml and secrets.toml files. This example uses chainlink-toml as the container name:

```
shell
docker run --platform linux/x8664/v8 --name chainlink-toml -v
/.chainlink:/chainlink -it -p 6688:6688 --add-host=host.docker.internal:host-
gateway smartcontract/chainlink:1.13.0 -config /chainlink/config.toml -
secrets /chainlink/secrets.toml node start
```

Test your node to verify that it works as intended. If you are using a VPS, open an SSH tunnel using `ssh -i $KEY $USER@$REMOTE-IP -L 6688:localhost:6688 -N`. Connect to the Operator UI in your browser at localhost:6688.

### Using multiple config files

You can use multiple config and secrets files. The config settings from each file are applied in the order that you specify when you run your node. Duplicated fields override values specified in earlier config files. This allows you to create a common config that applies to many nodes with specific configs for nodes that need unique configuration settings. Specifying multiple secrets files is invalid.

To specify multiple config files, add additional `-config` flags to the docker run command:

```
shell
docker run --platform linux/x8664/v8 --name chainlink -v /.chainlink:/chainlink
-it -p 6688:6688 --add-host=host.docker.internal:host-gateway
smartcontract/chainlink:1.13.0 -config /chainlink/config.toml -config
/chainlink/config2.toml -config /chainlink/config3.toml -secrets
/chainlink/secrets.toml node start
```

# node-versions.mdx:

```

section: nodeOperator
date: Last Modified
title: "Node Versions and Upgrades"
whatsnext: { "Running a Chainlink Node": "/chainlink-nodes/v1/running-a-
chainlink-node" }
metadata:
 title: "Node Versions and Release Notes"
 description: "Details about various node versions and how to migrate between
them."

```

```
import { Aside } from "@components"
```

You can find a list of release notes for Chainlink nodes in the smartcontractkit GitHub repository. Docker images are available in the Chainlink Docker hub.

Changes in v2.15.0 nodes - 2024-08-19

v2.15.0 release notes

Added

- Support for EIP-1559 transactions for Scroll
- Protocol-level support for preventing bid/ask variant violations in mercury
- Add VerboseLogging option to mercury. Off by default, can be enabled like so:

```
[Mercury]
VerboseLogging = true
```

#### Updated

- Dequeue minimum guaranteed upkeeps as a priority
- Rename the InBackupHealthReport to StartUpHealthReport and enable it for DB migrations as well. This will enable health report to be available during long start-up tasks such as DB backups and migrations:
  - VRFV2Plus coordinator and wrapper split contracts between L1 and L2 chains
  - Fix: Set LatestFinalizedBlock for finalized blocks saved by logpoller

Changes in v2.14.0 nodes - 2024-07-29

#### v2.14.0 release notes

##### Added

- Added Aptos Keystore to Core. This includes AptosKey which uses ED25519, Keystore, Relevant tests
- zkSync L1 GasPrice calculation

##### Updated

- Refactored the BlockHistoryEstimator check to prevent excessively bumping transactions. Check no longer waits for CheckInclusionBlocks to pass before assessing an attempt.
- Fixed a bug that would use the oldest blocks in the cached history instead of the latest to perform gas estimations.
- Updated L1 gas price calculations for Optimism Ecotone and Fjord upgrades
- Fixed local finality violation caused by an RPC lagging behind on latest finalized block.
- Fixed unreachable code bug which could result in stuck txns

##### DB Update

- CCIP capability specs migration

Changes in v2.13.0 nodes - 2024-07-01

#### v2.13.0 release notes

##### Breaking Change

- Removed the xdai ChainType config option. Moving forward, only gnosis can be used.

##### Added

- Added a mechanism to validate forwarders for OCR2 and fallback to EOA if necessary
- Added an auto-purge feature to the EVM TXM that identifies terminally stuck transactions either through a chain specific method or heuristic then purges them to unblock the nonce. Included four new toml configs under Transactions.AutoPurge to configure this new feature: Enabled, Threshold, MinAttempts, and DetectionApiUrl.
- Add option to include GasPriceSubunits pipeline to include gasPriceSubunits in

median ocr2 transmission. Use this only with the Starknet chain for now.

- Added config option `HeadTracker.FinalityTagBypass` to force `HeadTracker` to track blocks up to `FinalityDepth` even if `FinalityTagsEnabled = true`. This option is a temporary measure to address high CPU usage on chains with extremely large actual finality depth (gap between the current head and the latest finalized block).
- Added config option `HeadTracker.MaxAllowedFinalityDepth` maximum gap between current head to the latest finalized block that `HeadTracker` considers healthy.

#### Updated

- Fixed CPU usage issues caused by inefficiencies in `HeadTracker`. `HeadTracker`'s support of finality tags caused a drastic increase in the number of tracked blocks on the Arbitrum chain (from 50 to 12,000), which led to a 30% increase in CPU usage. The fix improves the data structure for tracking blocks and makes lookup more efficient. `BenchmarkHeadTrackerBackfill` shows a 40x time reduction.
- Fixed metric description on `mercurytransmitqueue` load

#### DB Updated

- Improve handling of Postgres connection settings and driver versions.

Changes in v2.12.0 nodes - 2024-06-05

#### v2.12.0 release notes

##### Added

- #12867 27d9413286 Added a new CLI command, `blocks find-lca`, which finds the latest block that is available in both the database and on the chain for the specified chain. Added a new CLI command, `node remove-blocks`, which removes all blocks and logs greater than or equal to the specified block number. #nops #added
- #12533 ccb8cd85fe Re-enable abandoned transaction tracker
- #12760 3f4573479c Enable configurable client error regexes for error classification. New toml configuration options for `[EVM.NodePool.Errors]` to pass regexes on `NonceTooLow`, `NonceTooHigh`, `ReplacementTransactionUnderpriced`, `LimitReached`, `TransactionAlreadyInMempool`, `TerminallyUnderpriced`, `InsufficientEth`, `TxFeeExceedsCap`, `L2FeeTooLow`, `L2FeeTooHigh`, `L2Full`, `TransactionAlreadyMined`, `Fatal`, and `ServiceUnavailable`.
- #12767 8db5ccfb39 Validate user email before asking for a password in the `chainlink` CLI.

##### Updated

- #12605 1d9dd466e2 `core/chains/evm/logpoller`: Stricter finality checks in `LogPoller`, to be more robust during rpc failover events
- #12595 e6d4814bda Move `JuelsPerFeeCoinCacheDuration` under `JuelsPerFeeCoinCache` struct in config. Rename `JuelsPerFeeCoinCacheDuration` to `updateInterval`. Add `stalenessAlertThreshold` to `JuelsPerFeeCoinCache` config. `StalenessAlertThreshold` cfg option has a default of 24 hours which means that it doesn't have to be set unless we want to override the duration after which a stale cache should start throwing errors.

#### Upcoming change in 2.13.0

- #12093 3f6d901fe6 The `xdai` `ChainType` has been renamed to `gnosis` to match the chain's new name. The old value is still supported but has been deprecated and will be removed in v2.13.0.

Changes in v2.11.0 nodes - 2024-05-01

#### v2.11.0 release notes

## Added

- Added a tx simulation feature to the chain client to enable testing for zk out-of-counter (OOC) errors
- Add the poolrpcnodehighestfinalizedblock metric that tracks the highest finalized block seen per RPC. If FinalityTagEnabled = true, a positive NodePool.FinalizedBlockPollInterval is needed to collect the metric. If the finality tag is not enabled, the metric is populated with a calculated latest finalized block based on the latest head and finality depth.

## Updated

- Moved JuelsPerFeeCoinCacheDuration under the JuelsPerFeeCoinCache struct in config. Rename JuelsPerFeeCoinCacheDuration to updateInterval. Add stalenessAlertThreshold to JuelsPerFeeCoinCache config. The StalenessAlertThreshold config option has a default of 24 hours which means that it doesn't have to be set unless we want to override the duration after which a stale cache should start throwing errors.
- Updated config for Polygon zkEVM chains.
- HeadTracker now respects the FinalityTagEnabled config option. If the flag is enabled, HeadTracker backfills blocks up to the latest finalized block provided by the corresponding RPC call. To address potential misconfigurations, HistoryDepth is now calculated from the latest finalized block instead of the head. Note that consumers like TXM and LogPoller do not fully use the Finality Tag yet.
- Change LimitTransfer gasLimit type from uint32 to uint64

## Upcoming changes in v2.13.0

- The xdai ChainType has been renamed to gnosis to match the chain's new name. The old value is still supported but has been deprecated and will be removed in v2.13.0.

## Changes in v2.10.0 nodes - 2024-04-08

### v2.10.0 release notes

## Added

- Gas bumping logic to the SuggestedPriceEstimator. The bumping mechanism for this estimator refetches the price from the RPC and adds a buffer on top using the greater of BumpPercent and BumpMin.
- Added a new configuration field named NodeIsSyncingEnabled for EVM.NodePool that will check on every reconnection to an RPC if it's syncing and should not be transitioned to Alive state. Disabled by default.
- Add preliminary support for "llo" job type (Data Streams V1)
- Add LogPrunePageSize parameter to the EVM configuration. This parameter controls the number of logs removed during prune phase in LogPoller. Default value is 0, which deletes all logs at once - exactly how it used to work, so it doesn't require any changes on the product's side.
- Add Juels Fee Per Coin data source caching for OCR2 Feeds. Cache is time based and is turned on by default with default cache refresh of 5 minutes. Cache can be configured through pluginconfig using "juelsPerFeeCoinCacheDuration" and "juelsPerFeeCoinCacheDisabled" tags. Duration tag accepts values between "30s" and "20m" with default of "0s" that is overridden on cache startup to 5 minutes.
- Add rebalancer support for feeds manager ocr2 plugins.

## Fixed

- P2P.V2 is required in configuration when either OCR or OCR2 are enabled. The node will fail to boot if P2P.V2 is not enabled.
- Removed unnecessary gas price warnings in gas estimators when EIP-1559 mode is enabled.



## Changed

- Minimum required version of Postgres is now  $\geq 12$ . Postgres 11 reached end-of-life (EOL) in November 2023. Added a new version check that will prevent Chainlink from running on EOL'd Postgres. If you are running Postgres  $\leq 11$  you must upgrade to the latest version. The check can be forcibly overridden by setting `SKIPPGVERSIONCHECK=true`.
- Updated the `LimitDefault` and `LimitMax` configs types to `uint64`

Changes in v2.9.1 nodes - 2024-03-07

## v2.9.1 release notes

### Changed

- `ethcall` RPC requests are now sent with both input and data fields to increase compatibility with servers that recognize only one.
- `GasEstimator` will now include Type `0x3` (Blob) transactions in the gas calculations to estimate it more accurately.

Changes in v2.9.0 nodes - 2024-02-22

## v2.9.0 release notes

### Added

- The chainlink health CLI command and HTML `/health` endpoint to provide human-readable views of the underlying JSON health data.
- New job type stream to represent streamspecs. This job type is not yet used anywhere but will be required for Data Streams V1.
- Environment variables `CLMEDIANENV`, `CLSOLANAENV`, and `CLSTARKNETENV` for setting environment variables in LOOP Plugins with an `.env` file.

```
echo "Foo=Bar" >> median.env
echo "Baz=Val" >> median.env
CLMEDIANENV="median.env"
```

### Fixed

- Fixed the encoding used for transactions when resending in batches.

### Removed

- `P2P.V1` is no longer supported and must not be set in the TOML configuration in order to boot. Use `P2P.V2` instead. If you are using both, V1 can simply be removed.
- Removed `TelemetryIngress.URL` and `TelemetryIngress.ServerPubKey` from TOML configuration, these fields are replaced by `[[TelemetryIngress.Endpoints]]`:

```
toml
[[TelemetryIngress.Endpoints]]
Network = '...' e.g. EVM, Solana, Starknet, Cosmos
ChainID = '...' e.g. 1, 5, devnet, mainnet-beta
URL = '...'
ServerPubKey = '...'
```

Changes in v2.8.0 nodes - 2024-01-24

## v2.8.0 release notes

### Added

- Added distributed tracing in the OpenTelemetry trace format to the node, currently focused at the LOOPP Plugin development effort. This includes a new set of Tracing TOML configurations. The default for collecting traces is off - you must explicitly enable traces and setup a valid OpenTelemetry collector. Refer to the README for more details.

- Added a new, optional WebServer authentication option that supports LDAP as a user identity provider. This enables user login access and user roles to be managed and provisioned via a centralized remote server that supports the LDAP protocol, which can be helpful when running multiple nodes. See the documentation for more information and config setup instructions. There is a new [WebServer].AuthenticationMethod config option, when set to ldap requires the new [WebServer.LDAP] config section to be defined, see the reference docs/core.toml.

- New prom metrics for mercury transmit queue:  
mercurytransmitqueuedeleteerrorcount  
mercurytransmitqueueinserterrorcount  
mercurytransmitqueuepusherrorcount  
Nops should consider alerting on these.

- Mercury now implements a local cache for fetching prices for fees, which ought to reduce latency and load on the mercury server, as well as increasing performance. It is enabled by default and can be configured with the following new config variables:

#### [Mercury]

Mercury.Cache controls settings for the price retrieval cache querying a mercury server

##### [Mercury.Cache]

LatestReportTTL controls how "stale" we will allow a price to be e.g. if set to 1s, a new price will always be fetched if the last result was from 1 second ago or older.

#

Another way of looking at it is such: the cache will never return a price that was queried from now-LatestReportTTL or before.

#

Setting to zero disables caching entirely.

LatestReportTTL = "1s" Default

MaxStaleAge is that maximum amount of time that a value can be stale before it is deleted from the cache (a form of garbage collection).

#

This should generally be set to something much larger than LatestReportTTL. Setting to zero disables garbage collection.

MaxStaleAge = "1h" Default

LatestReportDeadline controls how long to wait for a response from the mercury server before retrying. Setting this to zero will wait indefinitely.

LatestReportDeadline = "5s" Default

- New prom metrics for the mercury cache:  
mercurycachefetchfailurecount  
mercurycachehitcount  
mercurycachewaitcount  
mercurycachemisscount

- Added new EVM.OCR TOML config fields DeltaCOverride and DeltaCJitterOverride for overriding the config DeltaC.

- Mercury v0.2 has improved consensus around current block that uses the most recent 5 blocks instead of only the latest one

- Two new prom metrics for mercury, nops should consider adding alerting on these:

- mercuryinsufficientblockcount
- mercuryzeroblockcount

#### Changed

- PromReporter no longer directly reads txm related status from the db, and instead uses the txStore API.

- L2Suggested mode is now called SuggestedPrice

- Console logs will now escape (non-whitespace) control characters

- Following EVM Pool metrics were renamed:

- evmpoolrpcnodestates & rarr; multinodestates
- evmpoolrpcnodenumtransitionstoalive & rarr; poolrpcnodenumtransitionstoalive
- evmpoolrpcnodenumtransitionstoinsync & rarr;

poolrpcnodenumtransitionstoinsync

- evmpoolrpcnodenumtransitionstooutofsync & rarr;

poolrpcnodenumtransitionstooutofsync

- evmpoolrpcnodenumtransitionstounreachable & rarr;

poolrpcnodenumtransitionstounreachable

- evmpoolrpcnodenumtransitionstoinvalidchainid & rarr;

poolrpcnodenumtransitionstoinvalidchainid

- evmpoolrpcnodenumtransitionstounusable & rarr;

poolrpcnodenumtransitionstounusable

- evmpoolrpcnodehighestseenblock & rarr; poolrpcnodehighestseenblock

- evmpoolrpcnodenumseenblocks & rarr; poolrpcnodenumseenblocks

- evmpoolrpcnodepolltotal & rarr; poolrpcnodepolltotal

- evmpoolrpcnodepollsfailed & rarr; poolrpcnodepollsfailed

- evmpoolrpcnodepollssuccess & rarr; poolrpcnodepollssuccess

#### Removed

- Removed Optimism2 as a supported gas estimator mode

#### Fixed

- Corrected Ethereum Sepolia LinkContractAddress to

0x779877A7B0D9E8603169DdbD7836e478b4624789

- Fixed a bug that caused the Telemetry Manager to report incorrect health

#### Upcoming required configuration changes

#### Starting in v2.9.0:

- TelemetryIngress.URL and TelemetryIngress.ServerPubKey will no longer be allowed. Any TOML configuration that sets this fields will prevent the node from booting. These fields will be replaced by [[TelemetryIngress.Endpoints]]

- P2P.V1 will no longer be supported and must not be set in TOML configuration in order to boot. Use P2P.V2 instead. If you are using both, V1 can simply be removed.

#### Changes in v2.7.2 nodes - 2023-12-14

#### v2.7.2 release notes

#### Fixed

- Fixed a bug that caused nodes without OCR or OCR2 enabled to fail config validation if P2P.V2 was not explicitly disabled. With this fix, NOPs will not have to make changes to their config.

#### Changes in v2.7.1 nodes

## v2.7.1 release notes

### Fixed

- Fixed a bug that causes the node to shutdown if all configured RPC's are unreachable during startup.

### Changes in v2.7.0 nodes

## v2.7.0 release notes

### Added

- Added new configuration field named LeaseDuration for EVM.NodePool that will periodically check if internal subscriptions are connected to the "best" (as defined by the SelectionMode) node and switch to it if necessary. Setting this value to 0s will disable this feature.
- Added multichain telemetry support. Each network/chainID pair must be configured using the new fields:

```
toml
[[TelemetryIngress.Endpoints]]
Network = '...' e.g. EVM, Solana, Starknet, Cosmos
ChainID = '...' e.g. 1, 5, devnet, mainnet-beta
URL = '...'
ServerPubKey = '...'
```

These will eventually replace TelemetryIngress.URL and TelemetryIngress.ServerPubKey. Setting TelemetryIngress.URL and TelemetryIngress.ServerPubKey alongside [[TelemetryIngress.Endpoints]] will prevent the node from booting. Only one way of configuring telemetry endpoints is supported.

- Added bridgename label to pipelinetaskstotalfinished prometheus metric. This should make it easier to see directly what bridge was failing out from the CL NODE perspective.
- LogPoller will now use finality tags to dynamically determine finality on evm chains if UseFinalityTags=true, rather than the fixed FinalityDepth specified in toml config

### Changed

- P2P.V1 is now disabled (Enabled = false) by default. It must be explicitly enabled with true to be used. However, it is deprecated and will be removed in the future.
- P2P.V2 is now enabled (Enabled = true) by default.

### Upcoming required configuration changes

#### Starting in v2.9.0:

- TelemetryIngress.URL and TelemetryIngress.ServerPubKey will no longer be allowed. Any TOML configuration that sets this fields will prevent the node from booting. These fields will be replaced by [[TelemetryIngress.Endpoints]]
- P2P.V1 will no longer be supported and must not be set in TOML configuration in order to boot. Use P2P.V2 instead. If you are using both, V1 can simply be removed.

### Removed

- Removed the ability to set a next nonce value for an address through CLI

## Changes in v2.6.0 nodes

### v2.6.0 release notes

#### Added

- Simple password use in production builds is no longer allowed. Nodes with this configuration will not boot and will not pass config validation.
- Helper migrations function for injecting env vars into goose migrations. This was done to inject chainID into evm chain id not null in specs migrations.
- OCR2 jobs now support querying the state contract for configurations if it has been deployed. This can help on chains such as BSC which manage state bloat by arbitrarily deleting logs older than a certain date. In this case, if logs are missing we will query the contract directly and retrieve the latest config from chain state. Chainlink Nodes will perform no extra RPC calls unless the job spec has this feature explicitly enabled. On chains that require this, nops may see an increase in RPC calls. This can be enabled for OCR2 jobs by specifying ConfigContractAddress in the relay config TOML.

#### Removed

- Removed support for sending telemetry to the deprecated Explorer service. All nodes will have to remove Explorer related keys from TOML configuration and env vars.
- Removed default evmChainID logic where evmChainID was implicitly injected into the jobspecs based on node EVM chainID toml configuration. All newly created jobs that have the evmChainID field must explicitly define evmChainID in the job spec.
- Removed keyset migration that migrated v1 keys to v2 keys. All keys should've been migrated by now, and we don't permit creation of new v1 keys anymore

All nodes must remove the following secret configurations:

- Explorer.AccessKey
- Explorer.Secret

All nodes must remove the following configuration field: ExplorerURL

#### Fixed

- Unauthenticated users executing CLI commands previously generated a confusing error log, which is now removed:  
[ERROR] Error in transaction, rolling back: session missing or expired, please login again pg/transaction.go:118
- Fixed a bug that was preventing job runs to be displayed when the job chainID was disabled.
- chainlink txs evm create returns a transaction hash for the attempted transaction in the CLI. Previously only the sender, recipient and unstated state were returned.
- Fixed a bug where evmChainId is requested instead of id or evm-chain-id in CLI error verbatim
- Fixed a bug that would cause the node to shut down while performing backup
- Fixed health checker to include more services in the prometheus health metric and HTTP /health endpoint

## Changes in v2.5.0 nodes

### v2.5.0 release notes

#### Upcoming Required Configuration Change

- Starting in 2.6.0, chainlink nodes will no longer allow insecure configuration for production builds. Any TOML configuration that sets the following line will

fail validation checks in node start or node validate:

AllowSimplePasswords=true

- To migrate on production builds, update the database password set in Database.URL to be 16 - 50 characters without leading or trailing whitespace. URI parsing rules apply to the chosen password - refer to RFC 3986 for special character escape rules.

Added

- Various Functions improvements

Changes in v2.4.0 nodes

v2.4.0 release notes

Fixed

- Updated v2/keys/evm and v2/keys/eth routes to return 400 and 404 status codes where appropriate. Previously 500s were returned when requested resources were not found or client requests could not be parsed.
- Fixed withdrawing ETH from CL node for EIP1559 enabled chains. Previously would error out unless validation was overridden with allowHigherAmounts.

Added

- Added the ability to specify and merge fields from multiple secrets files. Overrides of fields and keys are not allowed.

Upcoming Required Configuration Change

- Starting in 2.6.0, Chainlink nodes will no longer allow insecure configuration for production builds. Any TOML configuration that sets the following line will fail validation checks in node start or node validate:

AllowSimplePasswords=true

- To migrate on production builds, update the database password set in Database.URL to be 16 - 50 characters without leading or trailing whitespace. URI parsing rules apply to the chosen password - refer to RFC 3986 for special character escape rules.

Changes in v2.3.0 nodes

v2.3.0 release notes

Added

- Add a new field called Order (range from 1 to 100) to EVM.Nodes that is used for the PriorityLevel node selector and also as a tie-breaker for HighestHead and TotalDifficulty. Order levels are considered in ascending order. If not defined it will default to Order = 100 (last level).
- Added new node selection mode called PriorityLevel for EVM, it is a tiered round-robin in ascending order of theOrder field. Example:

```
[EVM.NodePool]
SelectionMode = 'PriorityLevel'
```

```
[[EVM.Nodes]]
Name = '...'
WSURL = '...'
HTTPURL = '...'
Order = 5
```

- The config keys `WebServer.StartTimeout` and `WebServer.HTTPMaxSize`. These keys respectively set a timeout for the node server to start and set the max request size for HTTP requests. Previously these attributes were set by `JobPipeline.DefaultHTTPLimit/JobPipeline.DefaultHTTPTimeout`. To migrate to these new fields, set their values to be identical to `JobPipeline.DefaultHTTPLimit/JobPipeline.DefaultHTTPTimeout`.
- Low latency oracle jobs now support in-protocol block range guarantees. This is necessary in order to produce reports with block number ranges that do not overlap. It can now be guaranteed at the protocol level, so we can use local state instead of relying on an unreliable round-trip to the Mercury server.
- New settings `Evm.GasEstimator.LimitJobType.OCR2`, `OCR2.DefaultTransactionQueueDepth`, `OCR2.SimulateTransactions` for OCR2 jobs. These replace the settings `Evm.GasEstimator.LimitJobType.OCR`, `OCR.DefaultTransactionQueueDepth`, and `OCR.SimulateTransactions` for OCR.
- Add new config parameter to OCR and OCR2 named `TraceLogging` that enables trace logging of OCR and OCR2 jobs, previously this behavior was controlled from the `P2P.TraceLogging` parameter. To maintain the same behavior set `OCR.TraceLogging` and `OCR2.TraceLogging` to the same value `P2P.TraceLogging` was set.
- Add two new config parameters `WebServer.ListenIP` and `WebServer.TLS.ListenIP` which allows binding Chainlink HTTP/HTTPS servers to a particular IP. The default is `'0.0.0.0'` which listens to all IP addresses (same behavior as before). Set to `'127.0.0.1'` to only allow connections from the local machine (this can be handy for local development).
- Add several new metrics for mercury feeds, related to WSRPC connections:
  - `mercurytransmittimeoutcount`
  - `mercurydialcount`
  - `mercurydialsuccesscount`
  - `mercurydialerrorcount`
  - `mercuryconnectionresetcount`

Node operators may wish to add alerting based around these metrics.

#### Fixed

- Fixed a bug in the nodes xxx list command that caused results to not be displayed correctly

#### Changed

- Assumption violations for `MaxFeePerGas >= BaseFeePerGas` and `MaxFeePerGas >= MaxPriorityFeePerGas` in EIP-1559 effective gas price calculation will now use a gas price if specified
- Config validation now enforces protection against duplicate chain ids and node fields per provided TOML file. Duplicates across multiple configuration files are still valid. If you have specified duplicate chain ids or nodes in a given configuration file, this change will error out of all node subcommands.
- Restricted scope of the `Evm.GasEstimator.LimitJobType.OCR`, `OCR.DefaultTransactionQueueDepth`, and `OCR.SimulateTransactions` settings so they apply only to OCR. Previously these settings would apply to OCR2 as well as OCR. You must use the OCR2 equivalents added above if you want your settings to apply to OCR2.

#### Removed

- Legacy chain types `Optimism` and `Optimism2`. `OptimismBedrock` is now used to

handle Optimism's special cases.

- Optimism Kovan configurations along with legacy error messages.

Changes in v2.2.0 nodes

v2.2.0 release notes

Added

- Added experimental support of runtime process isolation for Solana data feeds. This requires plugin binaries to be installed and configured using the CLSOLANACMD and CLMEDIANCMD environment variables. See the plugins/README.md for more information.

Fixed

- Fixed a bug which made it impossible to re-send the same transaction after abandoning it while manually changing the nonce.

Changed

- Set the default for EVM.GasEstimator.BumpTxDepth to EVM.Transactions.MaxInFlight.
- Bumped batch size defaults for EVM-specific configurations. If you are overriding any of these fields in your local config, consider if it is necessary:
  - LogBackfillBatchSize = 1000
  - RPCDefaultBatchSize = 250
  - GasEstimator.BatchSize = 25
- Dropped support for the Development Mode configuration. The CLDEV environment variable is now ignored on production builds.
- Updated the Docker image's PostgreSQL client (used for backups) to v15 in order to support PostgreSQL v15 servers.

Changes in v2.1.0 nodes

v2.1.0 release notes

- The chainlink db CLI commands now validate the TOML configuration and secrets before executing. These commands will report errors if any database-specific configuration is invalid.

Changes in v2.0.0 nodes

v2.0.0 release notes

Added

- Add OCR2 Plugin selection for FMS
- Added kebab case aliases for the following flags:
  - evm-chain-id alias for evmChainID in commands: chainlink blocks replay, chainlink forwarders track, chainlink keys ... chain
  - old-password alias for oldpassword in commands: chainlink keys ... import
  - new-password alias for newpassword in commands: chainlink keys ... export
  - new-role alias for newrole in commands: admin users chrole
  - set-next-nonce alias for setNextNonce in commands: chainlink keys ... chain

Changed

- TOML configuration and secrets are now scoped to chainlink node command rather than being global flags.
- TOML configuration validation has been moved from chainlink config validate to chainlink node validate.
- Move chainlink node {status,profile} to chainlink admin {status,profile}.



## Removed

- Configuration with legacy environment variables is no longer supported. TOML is required.

## Changes in v1.13.0 nodes

### v1.13.0 release notes

#### TOML Configuration

TOML configuration for Chainlink nodes is stable and recommended for mainnet deployments. TOML configuration will be the only supported configuration method starting with v2.0.0. Enable TOML configuration by specifying the `-config FILENAME.toml` flag with the path to your TOML file. Alternatively, you can specify the raw TOML config in the `CLCONFIG` environment variable. See the `CONFIG.md` and `SECRETS.md` on GitHub to learn more.

## Added

- Added support for sending OCR2 job specs to the Feeds Manager.
- Log poller filters are now saved in the database and restored on node startup to guard against missing logs during periods where services are temporarily unable to start.

## Updated

- TOML config: The environment variable `CLCONFIG` is always processed as the last configuration. This has the effect of being the final override for any values provided via configuration files.

## Changed

- The Feeds Manager is now enabled by default.

## Removed

- Terra is no longer supported.

## Changes in v1.12.0 nodes

### v1.12.0 release notes

## Added

- Prometheus gauge `mailboxloadpercent` for percent of "Mailbox" capacity used.
- New config option `JobPipeline.MaxSuccessfulRuns` caps the total number of saved completed runs per job. This is done in response to the `pipelineruns` table potentially becoming large, which can cause performance degradation. The default is set to 10,000. You can set it to 0 to disable run saving entirely. NOTE: This can only be configured via TOML and not with an environment variable.
- Prometheus gauge `vectorfeedsjobproposalcount` to track counts of job proposals partitioned by proposal status.
- Support for variable expression for the `minConfirmations` parameter on the `ethTx` task.

## Updated

- Removed the `KEEPERTURNFLAGENABLED` as all networks and nodes have switched this to true. The variable should be completely removed.
- Removed the `Keeper.UpkeepCheckGasPriceEnabled` config and the `KEEPERCHECKUPKEEPGASPRICEFEATUREENABLED` environment variable. This feature is deprecated and the variable should be completely removed.

## Fixed

- Fixed (SQLSTATE 42P18) error on Job Runs page, when attempting to view specific older or infrequently run jobs.
- The config dump subcommand was fixed to dump the correct config data. The P2P.V1.Enabled config logic incorrectly matched V2, by only setting explicit true values so that otherwise the default is used. The V1.Enabled default value is actually true already, and is now updated to only set explicit false values.

## Changes in v1.11.0 nodes

### v1.11.0 release notes

#### Added

- Added a new mode for the NODESELECTIONMODE environment variable. Use TotalDifficulty to select the node with the greatest total difficulty.
- Added the NODESYNCTHRESHOLD environment variable to ensure that live nodes do not lag too far behind.
- Added the BRIDGECACHETTL environment variable which caches bridge responses for a specified amount of time.
- Add the prometheus metrics labelled by bridge name for monitoring external adapter queries. The following metrics are included:
  - bridgelatencyseconds
  - bridgeerrorstotal
  - bridgecachehitstotal
  - bridgecacheerrorstotal
- Experimental: Added static configuration using TOML files as an alternative to the existing combination of environment variables and persisted database configurations. For this release, use TOML for configuration only on test networks. In the future with v2.0.0, TOML configuration will become the only supported configuration method. Enable TOML configuration by specifying the -config FILENAME.toml flag with the path to your TOML file. Alternatively, you can specify the raw TOML config in the CLCONFIG environment variable. See the CONFIG.md and SECRETS.md on GitHub to learn more.

#### Fixed

- Fixed a minor bug where Chainlink would not always resend all pending transactions when using multiple keys.

#### Updated

- NODENONEWHEADSTHRESHOLD=0 no longer requires NODESELECTIONMODE=RoundRobin.

## Changes in v1.10.0 nodes

### v1.10.0 release notes

#### Added

- Added an optional external logger AUDITLOGSFORWARDERURL: When set, this environment variable configures and enables an optional HTTP logger which is used specifically to send audit log events. Configure this logger with the following environment variables:
  - AUDITLOGGERFORWARDTOURL
  - AUDITLOGGERHEADERS
  - AUDITLOGGERJSONWRAPPERKEY
- Added automatic connectivity detection to automatically detect if there is a transaction propagation/connectivity issue and prevent bumping in these cases on EVM chains.

#### Changed

- The default maximum gas price on most networks is now effectively unlimited.
- Chainlink will bump as high as necessary to get a transaction included. Automatic connectivity detection prevents excessive bumping when there is a connectivity failure.
- If you want to change this, manually set the ETHMAXGASPRICEWEI environment variable.
- If the EVMChainID is not set explicitly in the job spec for a new OCR job, the field is now automatically added with a default chain ID.
- Old OCR jobs missing EVMChainID continue to run on any chain that the ETHCHAINID variable is set to (or the first chain if it is not set). This can be changed after a restart.
- Newly created OCR jobs run only on a single fixed chain and are unaffected by changes to ETHCHAINID after the job is added.
- It should no longer be possible to end up with multiple OCR jobs for a single contract running on the same chain. Only one job per contract per chain is allowed.
- If there are any existing duplicate jobs per contract and per chain, all but the jobs with the latest creation date will be pruned during the upgrade.

#### Fixed

- Fixed minor bug where Chainlink would attempt and fail to estimate a tip cap higher than the maximum configured gas price in EIP1559 mode. It now caps the tipcap to the max instead of erroring.
- Fixed a bug where it was impossible to remove ETHkeys that had extant transactions. Now, removing an ETH key will drop all associated data automatically, including past transactions.

#### Automatic connectivity detection

Chainlink will no longer bump excessively if the network is broken.

This feature only applies on EVM chains when using BlockHistoryEstimator (the most common case).

Chainlink will now try to automatically detect if there is a transaction propagation/connectivity issue and prevent bumping in these cases. This can help avoid the situation where RPC nodes are not propagating transactions for some reason (e.g., go-ethereum bug, networking issue ...etc) and Chainlink responds in a suboptimal way by bumping transactions to a very high price in an effort to get them mined. This can lead to unnecessary expense when the connectivity issue is resolved and the transactions are finally propagated into the mempool.

This feature is enabled by default with fairly conservative settings: if a transaction has been priced above the 90th percentile of the past 12 blocks, but still wants to bump due to not being mined, a connectivity/propagation issue is assumed and all further bumping will be prevented for this transaction. In this situation, Chainlink will start firing the blockhistoryestimatorconnectivityfailurecount prometheus counter and logging at critical level until the transaction is mined.

The default settings should work fine for most users. For advanced users, the values can be tweaked by changing BLOCKHISTORYESTIMATORCHECKINCLUSIONBLOCKS and BLOCKHISTORYESTIMATORCHECKINCLUSIONPERCENTILE.

To disable connectivity checking completely, set BLOCKHISTORYESTIMATORCHECKINCLUSIONBLOCKS=0.

#### Changes in v1.9.0 nodes

##### v1.9.0 release notes

- Added the length task and the lessthan task for jobs.
- Added the gasUnlimited parameter to the ethcall task.

- The Keys page in Operator UI includes several admin commands that were previously available only by using the keys eth chain commands:
  - Ability to abandon all current transactions: This is the same as the abandon CLI command. Previously it was necessary to edit the database directly to abandon transactions. This command makes it easier to resolve issues that require transactions to be abandoned.
  - Ability to enable/disable a key for a specific chain: This allows you to control keys on a per-chain basis.
  - Ability to manually set the nonce for a key. This gives you a way to set the next nonce for a specific key in the UI, which can be useful for debugging.

#### Changes in v1.8.1 nodes

##### v1.8.1 release notes

- Added several improvements for Arbitrum Nitro including a multi-dimensional gas model, with dynamic gas pricing and limits.
  - The new default estimator for Arbitrum networks uses the suggested gas price up to ETHMAXGASPRICEWEI with a 1000 gwei default value and an estimated gas limit up to ETHGASLIMITMAX with a 1,000,000,000 default.
  - Remove the GASESTIMATORMODE environment variable to use the new defaults.
  - ETHGASLIMITMAX to puts a maximum on the gas limit returned by the Arbitrum estimator.

#### Changes in v1.8.0 nodes

##### v1.8.0 release notes

##### Added

- Added the hexencode and base64encode tasks (pipeline). See the Hex Encode Task and Base64 Encode Task pages for examples.
- forwardingAllowed per job attribute to allow forwarding txs submitted by the job.
- Added Arbitrum Goerli configuration support.
- Added the NODESELECTIONMODE (EVM.NodePool.SelectionMode) environment variable, which controls node picking strategy. Supported values are:
  - HighestHead is the default mode, which picks a node that has the highest reported head number among other alive nodes. When several nodes have the same latest head number, the strategy sticks to the last used node. This mode requires NODENONEWHEADSTHRESHOLD to be configured, otherwise it will always use the first alive node.
  - RoundRobin mode iterates among available alive nodes. This was the default behavior prior to this release.
- New evm keys chain command. This can also be accessed at /v2/keys/evm/chain. This command has the following uses:
  - Manually set or reset a nonce: chainlink evm keys chain --address "0xEXAMPLE" --evmChainID 99 --setNextNonce 42
  - Enable a key for a particular chain: chainlink evm keys chain --address "0xEXAMPLE" --evmChainID 99 --setEnabled true
  - Disable a key for a particular chain: chainlink evm keys chain --address "0xEXAMPLE" --evmChainID 99 --setEnabled false
- It is now possible to use the same key across multiple chains.

##### Changed

- The chainlink admin users update command is replaced with chainlink admin users chrole. Only the role can be changed for a user.
- Keypath now supports paths with any depth, instead of limiting it to 2.
- Arbitrum chains are no longer restricted to only FixedPrice GASESTIMATORMODE.
- Updated Arbitrum Rinkeby & Mainnet configurations for Nitro.

##### Removed

- The setnextnonce local client command is removed and is replaced by the more general evm keys chain command client command.

Changes in v1.7.1 nodes

v1.7.1 release notes

Fixed

- Arbitrum Nitro client error support

Changes in v1.7.0 nodes

v1.7.0 release notes

Added

- p2pv2Bootstrappers is added as a new optional property of OCR1 job specs. The default can still be specified with the P2PV2BOOTSTRAPPERS environment variable.

- Added official support for the Sepolia testnet on Chainlink nodes.

- Added hexdecode task and the base64decode task (pipeline).

- Added support for the Besu execution client. Although Chainlink supports Besu, Besu itself has several issues that can make it unreliable. For additional context, see the following issues:

- [hyperledger/besu/issues/4212](#)
- [hyperledger/besu/issues/4192](#)
- [hyperledger/besu/issues/4114](#)

- Added Multi-user and Role Based Access Control functionality. This allows the root admin CLI user and additional admin users to create and assign tiers of role-based access to new users. These new API users are able to log in to the Operator UI independently and can each have specific roles tied to their account. There are four roles: admin, edit, run, and view.

- User management can be configured through the use of the new admin CLI command chainlink admin users. Be sure to run chainlink admin login. For example, a readonly user can be created with: chainlink admin users create --email=operator-ui-read-only@test.com --role=view.

- Updated documentation repo with a break down of actions to required role level

- Added gas limit control for individual job specs and individual job types. The following rule of precedence is applied:

1. The task-specific parameter gasLimit overrides anything else when specified. For example, the ethtx task has a gasLimit parameter that overrides the other defaults for this specific task.

1. The job-spec attribute gasLimit applies only to a specific job spec.

1. The job-type limits affect any jobs of the corresponding type. The following environment variables are available:

- ETHGASLIMITOCRJOBTYP
- ETHGASLIMITDRJOBTYP
- ETHGASLIMITVRFJOBTYP
- ETHGASLIMITFMJOBTYP
- ETHGASLIMITKEEPERJOBTYP

1. The global ETHGASLIMITDEFAULT (EVM.GasEstimator.LimitDefault) value is used only when the preceding rules are not set.

## Fixed

- Addressed a very rare bug where using multiple nodes with differently configured RPC tx fee caps could cause missed transactions. Ensure that your RPC nodes have no caps. For more information, see the performance and tuning guide.
- Improved handling of unknown transaction error types to make Chainlink more robust in certain cases on unsupported chains or RPC clients.

## Changes in v1.6.0 nodes

### v1.6.0 release notes

- Simplified password complexity requirements. All passwords used with Chainlink must meet the following requirements:
  - Must be 16 characters or more
  - Must not contain leading or trailing whitespace
  - User passwords must not contain the user's API email
- Simplified the Keepers job spec by removing the observation source from the required parameters.

## Changes in v1.5.0 nodes

### v1.5.0 release notes

- Chainlink will not boot if the Postgres database password is missing or insecure. Passwords must conform to the following rules:

- Must be longer than 12 characters
- Must comprise at least 3 of the following items:
  - Lowercase characters
  - Uppercase characters
  - Numbers
  - Symbols
- Must not comprise:
  - More than three identical consecutive characters
  - Leading or trailing whitespace (note that a trailing newline in the password file, if present, will be ignored)

For backward compatibility, you can bypass this check at your own risk by setting `SKIPDATABASEPASSWORDCOMPLEXITYCHECK=true`.

- The following environment variables are deprecated and will be removed in a future release. They are replaced by the following command line arguments:

- `INSECURESKIPVERIFY`: Replaced by the `--insecure-skip-verify` CLI argument
- `CLIENTNODEURL`: Replaced by the `--remote-node-url` URL CLI argument
- `ADMINCREDENTIALSFILE`: Replaced by the `--admin-credentials-file` FILE CLI argument

Run `./chainlink --help` to learn more about these CLI arguments.

- The Optimism2 `GASESTIMATORMODE` has been renamed to `L2Suggested`. The old name is still supported for now.
- The `p2pBootstrapPeers` property on OCR2 job specs has been renamed to `p2pv2Bootstrappers`.

## Added

- Added the `ETHUSEFORWARDERS` config option to enable transactions forwarding contracts.
- In the `directrequest` job pipeline, three new block variables are available:

- `$(jobRun.blockReceiptsRoot)` : the root of the receipts trie of the block (hash)
  - `$(jobRun.blockTransactionsRoot)` : the root of the transaction trie of the block (hash)
  - `$(jobRun.blockStateRoot)` : the root of the final state trie of the block (hash)
- ethtx tasks can now be configured to error if the transaction reverts onchain. You must set `failOnRevert=true` on the task to enable this behavior:

```
foo [type=ethtx failOnRevert=true ...]
```

The ethtx task now works as follows:

- If `minConfirmations == 0`, task always succeeds and `nil` is passed as output.
  - If `minConfirmations > 0`, the receipt is passed through as output.
  - If `minConfirmations > 0` and `failOnRevert=true` then the ethtx task will error on revert.
  - If `minConfirmations` is not set on the task, the chain default will be used which is usually 12 and always greater than 0.
- http task now allows specification of request headers. Use it like the following example:

```
foo [type=http headers=["\\\"X-Header-1\\\"\", \\\"value1\\\"\", \\\"X-Header-2\\\"\", \\\"value2\\\"\"]].
```

#### Fixed

- Fixed `maxunconfirmedage` metric. Previously this would incorrectly report the max time since the last rebroadcast, capping the upper limit to the `EthResender` interval. This now reports the correct value of total time elapsed since the first broadcast.
- Correctly handle the case where bumped gas would exceed the RPC node's configured maximum on Fantom. Note that node operators should check their Fantom RPC node configuration and remove the fee cap if there is one.
- Fixed handling of the Metis internal fee change.

#### Removed

- The Optimism OVM 1.0 `GASESTIMATORMODE` has been removed and the Optimism2 `GASESTIMATORMODE` has been renamed to `L2Suggested`.
- `MINOUTGOINGCONFIRMATIONS` has been removed and no longer has any effect. The `ETHFINALITYDEPTH` environment variable is now used as the default for ethtx confirmations instead. You can override this on a per-task basis by setting `minConfirmations` in the task definition. For example, `foo [type=ethtx minConfirmations=42 ...]`.

This setting might have a minor impact on performance for very high throughput chains. If you don't care about reporting task status in the UI, set `minConfirmations=0` in your job specs. For more details, see the [Optimizing EVM Performance](#) page.

#### Changes in v1.4.1 nodes

##### v1.4.1 release notes

- Added a fix to ensure that a failed `EthSubscribe` does not register `(rpc.ClientSubscription)(nil)`, which leads to a panic when unsubscribing.
- Fix parsing of float values on job specs.

## Changes in v1.4.0 nodes

### v1.4.0 release notes

- JSON parse tasks in TOML now support a custom separator parameter to substitute for the default ,.
- Slow SQL queries are now logged.
- Updated the block explorer URLs to include FTMScan and SnowTrace.
- Keeper upkeep order can now be shuffled.
- Several fixes. See the release notes for a full list of changes.

## Changes in v1.3.0 nodes

### v1.3.0 release notes

- Added disk rotating logs. See the Node Logging and LOGFILEMAXSIZE documentation for details.
- Added support for the force flag on the chainlink blocks replay CLI command. If set to true, already consumed logs that would otherwise be skipped will be rebroadcasted.
- Added a version compatibility check when using the CLI to login to a remote node. The bypass-version-check flag skips this check.
- Changed default locking mode to "dual". See the DATABASELOCKINGMODE documentation for details.
- Specifying multiple EVM RPC nodes with the same URL is no longer supported. If you see ERROR 0106evmnodeuniqueness.sql: failed to run SQL migration, you have multiple nodes specified with the same URL and you must fix this before proceeding with the upgrade.
- EIP-1559 is now enabled by default on the Ethereum Mainnet. See the EVMEIP1559DYNAMICFEES documentation for details.
- Added new Chainlink Automation feature that includes gas price in calls to checkUpkeep(). To enable the feature, set KEEPERCHECKUPKEEPGASPRICEFEATUREENABLED to true. Use this setting only on Polygon networks.

## Changes in v1.2.0 nodes

### v1.2.0 release notes

<Aside type="caution" title="Not for use on Solana or Terra">

Although this release provides SOLANAENABLED and TERRAENABLED environment variables, these are not intended for use on Solana or Terra mainnet blockchains.

</Aside>

#### Significant changes:

- Added support for the Nethermind Ethereum client.
- Added support for batch sending telemetry to the ingress server to improve performance.
- New environment variables: See the release notes for details.
- Removed the deleteuser CLI command.
- Removed the LOGTODISK environment variable.

See the v1.2.0 release notes for a complete list of changes and fixes.

## Changes in v1.1.0 nodes

### v1.1.0 release notes

The v1.1.0 release includes several substantial changes to the way you configure and operate Chainlink nodes:

- Legacy environment variables: Legacy environment variables are supported, but



they might be removed in future node versions. See the [Configuring Chainlink Nodes](#) page to learn how to migrate your nodes away from legacy environment variables and use the API, CLI, or GUI exclusively to administer chains and nodes.

- Full EIP1559 Support: Chainlink nodes include experimental support for submitting transactions using type 0x2 (EIP-1559) envelope. EIP-1559 mode is off by default, but can be enabled either globally or on a per-chain basis.
- New log level added:
  - [crit]: Critical level logs are more severe than [error] and require quick action from the node operator.
- Multichain support (Beta): Chainlink now supports connecting to multiple different EVM chains simultaneously. This is disabled by default. See the [v1.1.0 Changelog](#) for details.

With multiple chain support, eth node configuration is stored in the database.

The following environment variables are DEPRECATED:

- ETHURL
- ETHHTTPURL
- ETHSECONDARYURLS

Setting ETHURL will cause Chainlink to automatically overwrite the database records with the given ENV values every time Chainlink boots. This behavior is used mainly to ease the process of upgrading from older versions, and on subsequent runs (once your old settings have been written to the database) it is recommended to unset these ENV vars and use the API commands exclusively to administer chains and nodes.

If you wish to continue using these environment variables (as it used to work in 1.0.0 and below) you must ensure that the following are set:

- ETHCHAINID (mandatory)
- ETHURL (mandatory)
- ETHHTTPURL (optional)
- ETHSECONDARYURLS (optional)

If, instead, you wish to use the API/CLI/GUI to configure your chains and eth nodes (recommended) you must REMOVE the following environment variables:

- ETHURL
- ETHHTTPURL
- ETHSECONDARYURLS

This will cause Chainlink to use the database for its node configuration.

NOTE: ETHCHAINID does not need to be removed, since it now performs the additional duty of specifying the default chain in a multichain environment (if you leave ETHCHAINID unset, the default chain is simply the "first").

For more information on configuring your node, check the configuration variables in the docs.

Before you upgrade your nodes to v1.1.0, be aware of the following requirements:

- If you are upgrading from a previous version, you MUST first upgrade the node to at least v0.10.15.
- Always take a Database snapshot before you upgrade your Chainlink nodes. You must be able to roll the node back to a previous version in the event of an upgrade failure.

See the [v1.1.0 release notes](#) for a complete list of changes and fixes.

Changes in v1.0.0 and v1.0.1 nodes

v1.0.0 release notes  
v1.0.1 release notes

Before you upgrade your nodes to v1.0.0 or v1.0.1, be aware of the following requirements:

- If you are upgrading from a previous version, you MUST first upgrade the node to at least v0.10.15.
- Always take a Database snapshot before you upgrade your Chainlink nodes. You must be able to roll the node back to a previous version in the event of an upgrade failure.

# addresses.mdx:

```

section: nodeOperator
date: Last Modified
title: "Contract addresses"

```

import { Address } from "@components"

This page lists the operator factory addresses for different networks.

Ethereum

Mainnet

```
<Address
contractUrl="https://etherscan.io/address/0x3E64Cd889482443324F91bFA9c84fE72A511
f48A" />
```

Sepolia

```
<Address
contractUrl="https://sepolia.etherscan.io/address/0x447Fd5eC2D383091C22B8549cb23
1a3bAD6d3fAf" />
```

# forwarder.mdx:

```

section: nodeOperator
date: Last Modified
title: "Forwarder"

```

In the EVM world, externally-owned account transactions are confirmed sequentially. Chainlink nodes that use a single externally-owned account (EOA) per chain face several important challenges:

- Transactions are broadcast through a single FIFO queue, so the throughput is limited to a single lane. Low throughput is terrible for the overall user experience because concurrent clients must wait to have their requests fulfilled. When more clients make requests, the longer it takes for requests to be fulfilled.
- Transactions are not executed by priority. For example, consider a situation where there are two transactions in a row. The first transaction is a fulfillment of an API request to get the winner of the FIFA world cup 2022, and the second transaction is an ETH/USD price feed update used by multiple DeFi protocols. Relying on a single EOA forces the second transaction to be confirmed only after the first transaction is fulfilled. The first transaction is not as

time-sensitive, but it is still fulfilled first.

- A stuck transaction will cause all the subsequent transactions to remain pending. For example, if a transaction becomes stuck because the gas price is not set high enough, that transaction must be bumped or canceled before subsequent transactions can be fulfilled.

- As a workaround, some node operators deploy multiple Chainlink nodes per chain. While this allows them to handle different types of requests separately (one node for price feeds and another to fulfill API requests), this comes with an overhead in infrastructure and maintenance costs.

To solve these challenges, we introduced two major features that will allow node operators to set up different transaction-sending strategies more securely while lowering their infrastructure costs:

- Chainlink nodes support multiple EOAs.
- Forwarder contracts allow a node operator to manage multiple EOAs and make them look like a single address. If you use a web2 analogy, forwarder contracts act like a reverse proxy server where the user is served by the same address and does not see which server the traffic is coming from. To do so, nodes call the forward function on the forwarder contract.

Combining multiple EOAs and forwarder contracts allows greater flexibility and security in terms of design:

- Node operators can expand horizontally using multiple EOAs. They can deploy one or multiple forwarder contracts for these EOAs. The combination of EOAs and forwarders offers a lot of flexibility for setting up different pipelines for handling transactions.
- Node operators can support different job types (OCR, VRF, API request..Etc) on the same node, which reduces maintenance and infrastructure costs.
- Security-wise, forwarder contracts distinguish between owner accounts and authorized sender accounts. Authorized senders are hot wallets such as the EOAs of a Chainlink node. If a node is compromised, the owner is responsible for changing the authorized senders list.
- Node operators do not need to manually compile and deploy operator and forwarder contracts. They can deploy them directly from the operator factory by calling the `deployNewOperatorAndForwarder` function. From a design perspective, the owner of a forwarder contract is an operator contract. The owner of the operator contract is usually a more secure address with keys stored in a hardware wallet or protected by a multisig. Node operators can manage a set of forwarder contracts through an operator contract.

## API Reference

The forwarder contract inherits `AuthorizedReceiver.sol` and `ConfirmedOwnerWithProposal.sol`. Read the `Receiver` and `Ownership` API references to learn more.

## Methods

`typeAndVersion`

`solidity`

`function typeAndVersion() external pure virtual returns (string)`

The type and version of this contract.

## Return Values

Name	Type	Description
----	-----	-----
	string	Type and version string

forward

solidity

```
function forward(address to, bytes data) external
```

Forward a call to another contract.

Only callable by an authorized sender

Parameters

Name	Type	Description
to	address	address
data	bytes	to forward

ownerForward

solidity

```
function ownerForward(address to, bytes data) external
```

Forward a call to another contract.

Only callable by the owner

Parameters

Name	Type	Description
to	address	address
data	bytes	to forward

transferOwnershipWithMessage

solidity

```
function transferOwnershipWithMessage(address to, bytes message) external
```

Transfer ownership with instructions for recipient. Emit OwnershipTransferRequestedWithMessage event.

Parameters

Name	Type	Description
to	address	address proposed recipient of ownership
message	bytes	instructions for recipient upon accepting ownership

Events

OwnershipTransferRequestedWithMessage

solidity

```
event OwnershipTransferRequestedWithMessage(address from, address to, bytes message)
```

# operator.mdx:

---

section: nodeOperator

date: Last Modified  
title: "Operator"  
---

```
import { Aside } from "@components"
```

Oracles must deploy an onchain contract to handle requests made through the LINK token (Read Basic Request Model to learn more).

When the Basic Request model was introduced, node operators had to deploy the legacy Oracle contract. However, these come with some limitations, and soon, we introduced operator contracts.

```
<Aside type="note">
 <p>Node operators must use Operator.sol instead of Oracle.sol.</p>
</Aside>
```

In addition to replacing oracle contracts, operator contracts come with additional features that add more security and flexibility for node operators.

## Features

### Multi-word response

In the EVM architecture, a word is made up of 32 bytes. One limitation of the Oracle.sol contract is that it limits responses to requests to 32 bytes.

Operator.sol doesn't have the same limitation as it supports a response made of multiple EVM words.

### Factory deployment

To deploy an Oracle contract, each node operator has to manually compile and deploy Oracle.sol.

The vast number of Solidity versions and steps involved in verifying the contract made it difficult for a client to verify that the deployed contract had not been tampered with.

To fix this, node operators can use a factory to deploy an instance of the operator contract. Moreover, the factory exposes a getter for clients to check if it deployed a specific operator contract address.

### Distributing funds to multiple addresses

A common pain point of node operators is keeping their addresses funded. Operator's distributeFunds method allows node operators to fund multiple addresses in a single transaction.

### Flexibility and security

By using multiple externally-owned accounts (EOAs) on Chainlink nodes and forwarder contracts, node operators can set up different transaction-sending strategies.

```
<Aside type="note">
 <p>Read more about forwarders.</p>
</Aside>
```

As discussed in the forwarder contracts page:

- Chainlink nodes' EOAs are hot wallets that fulfill requests.
- These EOAs can be associated with one or multiple forwarder contracts. The forwarder's owner must whitelist them to call the forward function. One operator contract owns one or multiple forwarder contracts.
- Node operators manage their forwarder contracts through operator contracts.

They use a secure wallet such as hardware or a multisig wallet as the operator's owner account.

## API reference

The operator contract inherits `AuthorizedReceiver` and `ConfirmedOwnerWithProposal`. Read `AuthorizedReceiver` and `ConfirmedOwnerWithProposal` API references.

## Methods

### oracleRequest

<Aside type="note" title="Legacy">  
Use `operatorRequest` function instead.  
</Aside>

#### solidity

```
function oracleRequest(address sender, uint256 payment, bytes32 specId, address
callbackAddress, bytes4 callbackFunctionId, uint256 nonce, uint256 dataVersion,
bytes data) external
```

Creates the Chainlink request. This is backwards compatible API with `Oracle.sol` contracts, but the behavior changes because `callbackAddress` is assumed to be the same as the request sender.

## Parameters

Name	Type	Description
-----	-----	-----
sender	address	The sender of the request
payment	uint256	The amount of payment given (specified in wei)
specId	bytes32	The Job Specification ID
callbackAddress	address	The consumer of the request
callbackFunctionId	bytes4	The callback function ID for the response
nonce	uint256	The nonce sent by the requester
dataVersion	uint256	The specified data version
data	bytes	The extra request parameters

### operatorRequest

#### solidity

```
function operatorRequest(address sender, uint256 payment, bytes32 specId, bytes4
callbackFunctionId, uint256 nonce, uint256 dataVersion, bytes data) external
```

Creates the Chainlink request. Stores the hash of the params as the onchain commitment for the request. Emits `OracleRequest` event for the Chainlink node to detect.

## Parameters

Name	Type	Description
------	------	-------------

-----	-----	-----
sender	address	The sender of the request
payment	uint256	The amount of payment given (specified in wei)
specId	bytes32	The Job Specification ID
callbackFunctionId	bytes4	The callback function ID for the response
nonce	uint256	The nonce sent by the requester
dataVersion	uint256	The specified data version
data	bytes	The extra request parameters

## fulfillOracleRequest

<Aside type="note" title="Legacy">  
 Use fulfillOracleRequest2 function instead.  
 </Aside>

### solidity

```
function fulfillOracleRequest(bytes32 requestId, uint256 payment, address
callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes32 data)
external returns (bool)
```

Called by the Chainlink node to fulfill requests. Given params must hash back to the commitment stored from oracleRequest. Will call the callback address' callback function without bubbling up error checking in a require so that the node can get paid. Emits OracleResponse event.

### Parameters

Name	Type	Description
-----	-----	-----
requestId	bytes32	The fulfillment request ID that must match the requester's
payment	uint256	The payment amount that will be released for the oracle (specified in wei)
callbackAddress	address	The callback address to call for fulfillment
callbackFunctionId	bytes4	The callback function ID to use for fulfillment
expiration	uint256	The expiration that the node should respond by before the requester can cancel
data	bytes32	The data to return to the consuming contract

### Return values

Name	Type	Description
----	-----	-----
	bool	Status if the external call was successful

## fulfillOracleRequest2

### solidity

```
function fulfillOracleRequest2(bytes32 requestId, uint256 payment, address
```

callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes data)  
external returns (bool)

Called by the Chainlink node to fulfill requests with multi-word support. Given params must hash back to the commitment stored from oracleRequest. Will call the callback address' callback function without bubbling up error checking in a require so that the node can get paid. Emits OracleResponse event.

#### Parameters

Name	Type	Description
-----	-----	-----
requestId	bytes32	The fulfillment request ID that must match the requester's
payment	uint256	The payment amount that will be released for the oracle (specified in wei)
callbackAddress	address	The callback address to call for fulfillment
callbackFunctionId	bytes4	The callback function ID to use for fulfillment
expiration	uint256	The expiration that the node should respond by before the requester can cancel
data	bytes	The data to return to the consuming contract

#### Return values

Name	Type	Description
----	----	-----
	bool	Status if the external call was successful

#### transferOwnableContracts

##### solidity

```
function transferOwnableContracts(address[] ownable, address newOwner) external
```

Transfer the ownership of ownable contracts. This is primarily intended for authorized forwarders but could possibly be extended to work with future contracts.

#### Parameters

Name	Type	Description
-----	-----	-----
ownable	address[]	list of addresses to transfer
newOwner	address	address to transfer ownership to

#### acceptOwnableContracts

##### solidity

```
function acceptOwnableContracts(address[] ownable) public
```

Accept the ownership of an ownable contract. This is primarily intended for authorized forwarders but could possibly be extended to work with future contracts. Emits OwnableContractAccepted event.

Must be the pending owner on the contract

#### Parameters



Name	Type	Description
ownable	address[]	list of addresses of Ownable contracts to accept

setAuthorizedSendersOn

solidity

```
function setAuthorizedSendersOn(address[] targets, address[] senders) public
```

Sets the fulfillment permission for senders on targets. Emits TargetsUpdatedAuthorizedSenders event.

Parameters

Name	Type	Description
targets	address[]	The addresses to set permissions on
senders	address[]	The addresses that are allowed to send updates

acceptAuthorizedReceivers

solidity

```
function acceptAuthorizedReceivers(address[] targets, address[] senders)
external
```

Accepts ownership of ownable contracts and then immediately sets the authorized sender list on each of the newly owned contracts. This is primarily intended for authorized forwarders but could possibly be extended to work with future contracts.

Parameters

Name	Type	Description
targets	address[]	The addresses to set permissions on
senders	address[]	The addresses that are allowed to send updates

withdraw

solidity

```
function withdraw(address recipient, uint256 amount) external
```

Allows the node operator to withdraw earned LINK to a given address recipient.

The owner of the contract can be another wallet and does not have to be a Chainlink node

Parameters

Name	Type	Description
recipient	address	The address to send the LINK token to
amount	uint256	The amount to send (specified in wei)

withdrawable

solidity

```
function withdrawable() external view returns (uint256)
```

Displays the amount of LINK that is available for the node operator to withdraw.

We use 1 in place of 0 in storage

Return values

Name	Type	Description
-----	-----	-----
	uint256	The amount of withdrawable LINK on the contract

ownerForward

solidity

function ownerForward(address to, bytes data) external

Forward a call to another contract.

Only callable by the owner

Parameters

Name	Type	Description
-----	-----	-----
to	address	address
data	bytes	to forward

ownerTransferAndCall

solidity

function ownerTransferAndCall(address to, uint256 value, bytes data) external  
returns (bool success)

Interact with other LinkTokenReceiver contracts by calling transferAndCall.

Parameters

Name	Type	Description
-----	-----	-----
to	address	The address to transfer to.
value	uint256	The amount to be transferred.
data	bytes	The extra data to be passed to the receiving contract.

Return values

Name	Type	Description
-----	-----	-----
success	bool	bool

distributeFunds

solidity

function distributeFunds(address payable[] receivers, uint256[] amounts)  
external payable

Distribute funds to multiple addresses using ETH sent to this payable function.

Array length must be equal, ETH sent must equal the sum of amounts. A malicious receiver could cause the distribution to revert, in which case it is expected that the address is removed from the list.

Parameters

Name	Type	Description
receivers	address payable[]	list of addresses
amounts	uint256[]	list of amounts

cancelOracleRequest

solidity

```
function cancelOracleRequest(bytes32 requestId, uint256 payment, bytes4
callbackFunc, uint256 expiration) external
```

Allows recipient to cancel requests sent to this oracle contract. Will transfer the LINK sent for the request back to the recipient address. Given params must hash to a commitment stored on the contract in order for the request to be valid. Emits CancelOracleRequest event.

Parameters

Name	Type	Description
requestId	bytes32	The request ID
payment	uint256	The amount of payment given (specified in wei)
callbackFunc	bytes4	The requester's specified callback function selector
expiration	uint256	The time of the expiration for the request

cancelOracleRequestByRequester

solidity

```
function cancelOracleRequestByRequester(uint256 nonce, uint256 payment, bytes4
callbackFunc, uint256 expiration) external
```

Allows requester to cancel requests sent to this oracle contract. Will transfer the LINK sent for the request back to the recipient address. Given params must hash to a commitment stored on the contract in order for the request to be valid. Emits CancelOracleRequest event.

Parameters

Name	Type	Description
nonce	uint256	The nonce used to generate the request ID
payment	uint256	The amount of payment given (specified in wei)
callbackFunc	bytes4	The requester's specified callback function selector
expiration	uint256	The time of the expiration for the request

getChainlinkToken

solidity

```
function getChainlinkToken() public view returns (address)
```

Returns the address of the LINK token

This is the public implementation for `chainlinkTokenAddress`, which is an internal method of the `ChainlinkClient` contract.

Events

`OracleRequest`

```
solidity
event OracleRequest(bytes32 specId, address requester, bytes32 requestId,
uint256 payment, address callbackAddr, bytes4 callbackFunctionId, uint256
cancelExpiration, uint256 dataVersion, bytes data)
```

`CancelOracleRequest`

```
solidity
event CancelOracleRequest(bytes32 requestId)
```

`OracleResponse`

```
solidity
event OracleResponse(bytes32 requestId)
```

`OwnableContractAccepted`

```
solidity
event OwnableContractAccepted(address acceptedContract)
```

`TargetsUpdatedAuthorizedSenders`

```
solidity
event TargetsUpdatedAuthorizedSenders(address[] targets, address[] senders,
address changedBy)
```

# operatorfactory.mdx:

```

section: nodeOperator
date: Last Modified
title: "Operator Factory"

```

The factory design pattern is a well know programming pattern: Rather than compiling and creating instances of a contract yourself, the factory does it for you.

As explained in the operator guide, the `OperatorFactory` contract comes with these benefits:

- Node operators do not need to manually compile and deploy operator or/and forwarder contracts. They can deploy them directly from the factory. See `thedeploynewoperator`, `deploynewforwarder`, and `deploynewoperatorandforwarder` functions.
- Clients can verify if the factory deployed a given contract. See the `created` function.

## API Reference

### Methods

#### typeAndVersion

solidity

function typeAndVersion() external pure virtual returns (string)

The type and version of this contract.

#### Return values

Name	Type	Description
----	-----	-----
	string	Type and version string

#### deployNewOperator

solidity

function deployNewOperator() external returns (address)

Creates a new operator contract with the msg.sender as owner. Emits OperatorCreated event.

#### deployNewOperatorAndForwarder

solidity

function deployNewOperatorAndForwarder() external returns (address, address)

Creates a new operator contract with the msg.sender as the owner and a new forwarder with the operator as the owner. Emits:

- OperatorCreated event.
- AuthorizedForwarderCreated event.

#### deployNewForwarder

solidity

function deployNewForwarder() external returns (address)

Creates a new forwarder contract with the msg.sender as owner. Emits AuthorizedForwarderCreated event.

#### deployNewForwarderAndTransferOwnership

solidity

function deployNewForwarderAndTransferOwnership(address to, bytes message) external returns (address)

Creates a new forwarder contract with the msg.sender as owner. Emits AuthorizedForwarderCreated event.

#### created

solidity

function created(address query) external view returns (bool)

Indicates whether this factory deployed an address.

## Events

### OperatorCreated

solidity

```
event OperatorCreated(address operator, address owner, address sender)
```

### AuthorizedForwarderCreated

solidity

```
event AuthorizedForwarderCreated(address forwarder, address owner, address sender)
```

# ownership.mdx:

---

section: nodeOperator

date: Last Modified

title: "Ownership"

---

ConfirmedOwnerWithProposal is inherited by operator and forwarder contracts. It contains helpers for basic contract ownership.

## API reference

### Methods

#### transferOwnership

solidity

```
function transferOwnership(address to) public
```

Allows an owner to begin transferring ownership to a new address. Emits an OwnershipTransferRequested event.

#### acceptOwnership

solidity

```
function acceptOwnership() external
```

Allows an ownership transfer to be completed by the recipient. Emits an OwnershipTransferred event.

#### owner

solidity

```
function owner() public view returns (address)
```

Get the current owner.

## Events

### OwnershipTransferRequested

solidity

event OwnershipTransferRequested(address from, address to)

OwnershipTransferred

solidity

event OwnershipTransferred(address from, address to)

# receiver.mdx:

---

section: nodeOperator

date: Last Modified

title: "Receiver"

---

import { Aside } from "@components"

AuthorizedReceiver is an abstract contract inherited by operator and forwarder contracts.

<Aside type="note">

Calling setAuthorizedSenders has a different effect depending if it is called from an operator or a forwarder contract:

- Owners of forwarder contracts allow authorized senders to call forward.
- Owners of operator contracts allow authorized senders to call the fulfillOracleRequest and fulfillOracleRequest2 methods.

</Aside>

API reference

Methods

setAuthorizedSenders

solidity

function setAuthorizedSenders(address[] senders) external

Sets the fulfillment permission for a given node. Use true to allow, false to disallow.

Emits an AuthorizedSendersChanged event.

Parameters

Name	Type	Description
senders	address[]	The addresses of the authorized Chainlink node

getAuthorizedSenders

solidity

function getAuthorizedSenders() external view returns (address[])

Retrieve a list of authorized senders.

Return values

Name	Type	Description
------	------	-------------

	----		-----		-----	
			address[]		array of addresses	

isAuthorizedSender

solidity

function isAuthorizedSender(address sender) public view returns (bool)

Use this to check if a node is authorized to fulfill requests.

Parameters

	Name		Type		Description	
	-----		-----		-----	
	sender		address		The address of the Chainlink node	

Return values

	Name		Type		Description	
	----		-----		-----	
			bool		The authorization status of the node	

Events

AuthorizedSendersChanged

solidity

event AuthorizedSendersChanged(address[] senders, address changedBy)

# building-external-initiators.mdx:

---

section: nodeOperator

date: Last Modified

title: "Building External Initiators"

whatsnext:

{ "Adding External Initiators to Nodes": "/chainlink-nodes/external-initiators/external-initiators-in-nodes" }

---

import { Aside } from "@components"

An external initiator can trigger a run for any webhook job that it has been linked to.

The URL for triggering a run is such:

shell

curl -X POST -H "Content-Type: application/json" --data '{"myKey": "myVal"}'  
http://localhost:6688/v2/jobs/<job external UUID>/runs

You will need to specify two headers:

1. "X-Chainlink-EA-AccessKey"

1. "X-Chainlink-EA-Secret"

JSON jobs (REMOVED)

<Aside type="caution" title="v1 Jobs are removed">

The initiators for v1 Jobs are removed for Chainlink nodes running version



1.0.0 and later. Use v2 job types instead.

<br />

See the v2 jobs migration page to learn how to migrate to v2 jobs.

</Aside>

We will be using the <a href="https://github.com/smartcontractkit/external-initiator">Chainlink external initiator</a> repo for reference. You can see some examples of existing initiators in the <a href="https://github.com/smartcontractkit/external-initiator/tree/master/blockchain">blockchain</a> folder.

External initiators are simple web initiators that can be activated by any job instead of just one. To set one up, you need to have a service similar to an external adapter that sends an HTTPPost message runs API call to your chainlink node service. Here is a sample URL for a web job could look like:

shell

```
curl -b cookiefile -X POST -H "Content-Type: application/json" --data
'{"myKey":"myVal"}' http://localhost:6688/v2/jobs/%s/runs
```

Where %s is the jobId.

External initiators make the same API call, with 2 added headers:

1. "X-Chainlink-EA-AccessKey"
1. "X-Chainlink-EA-Secret"

These are keys generated when you register your external initiator with your node.

Triggering a run through an external initiator is as simple as making this API call to your node. All jobs with this EI configured will then be kicked off in this way. A simple external initiator in psedo code could look like this:

text

```
while(True):
 sendapicallwithexternalinitiatoraccesskeyheaders()
 sleep(4)
```

And have this job run on the same machine as your node.

# external-initiators-in-nodes.mdx:

---

```
section: nodeOperator
date: Last Modified
title: "Adding External Initiators to Nodes"

```

```
import { Aside } from "@components"
```

<Aside type="note">

External initiators are disabled on nodes by default. Set the FEATUREEXTERNALINITIATORS=true configuration variable to enable this feature.

</Aside>

Creating an external initiator

To create an external initiator you must use the remote API. You can do this

yourself, like so:

```
{/ prettier-ignore /}
text
POST http://<your chainlink node>/v2/externalinitiators -d <PAYLOAD>
```

where payload is a JSON blob that contains:

```
{/ prettier-ignore /}
json
{
 "name": <MANDATORY UNIQUE NAME>,
 "url": <OPTIONAL EXTERNAL INITIATOR URL>
}
```

If a URL is provided, Chainlink will notify this URL of added and deleted jobs that can be triggered by this external initiator. This allows the external initiator to program in certain actions e.g. subscribing/unsubscribing to logs based on the job, etc.

On creation:

```
{/ prettier-ignore /}
text
POST <URL> -d {"jobId": <job external UUID>, "type": <name of external
initiator>, "params": <optional arbitrary JSON specified at job creation time>}
```

On deletion:

```
{/ prettier-ignore /}
text
DELETE <URL>/<job external UUID>
```

You can use the chainlink client for convenience to access this API.

Enter the Chainlink nodes CLI and run the following command

```
{/ prettier-ignore /}
shell
chainlink initiators create <NAME> <URL>
```

NAME: The name you want to use for your external initiator.

URL: The URL of your jobs endpoint. ie: http://172.17.0.1:8080/jobs

This will give you the environment variables you need to run your external initiator. Copy the output. It will look something like this:

```
â ‘ einame â ‘ http://localhost:8080/jobs â ‘ a4846e85727e46b48889c6e28b555696
â ‘ dnNfNhiICTm1o6l+hGJVfCtRSSuDfZbj1V04BkZG3E+b96lminE7yQHj2KALMAIk â ‘
iWt64+Q9benOf5JuGwJtQnbByN9rtHwSLElOVpHVTvGTP5Zb2Guwzy6w3wflwyYt â ‘
56m38YkeCymYU0kr4Yg6x3e98CyAu+37y2+kM02AL9lRMjA3hRA1ejFdG9UfFCAE
```

Be sure to save these values, since the secrets cannot be shown again.

You now can use einame as an initiator in your jobspec.

Set a new .env file, and add the respective values

```
text
EIDATABASEURL=postgresql://$USERNAME:$PASSWORD@$SERVER:$PORT/$DATABASE
EICHAINLINKURL=http://localhost:6688
EIICACCESSKEY=<INSERT KEY>
EIICSECRET=<INSERT KEY>
EICIACCESSKEY=<INSERT KEY>
EICISECRET=<INSERT KEY>
```

At the time of writing, the output should be in order. For example, in from the output above, EIICACCESSKEY=a4846e85727e46b48889c6e28b555696 and so on.

Start your EI.

Whatever code you used to run your external initiator, pass it the new headers created for the access headers, and then start your service. An easy way to do this is by having it read from the .env file you just created. Check out the [Conflux External initiator](https://github.com/Conflux-Network-Global/demo-cfx-chainlink) for an example.

You'll want to test that your job is running properly. Meeting the criteria of your EI and then checking to see if a sample job kicks off is the best way to test this.

To try a real-life example, feel free to follow along with the [Conflux EI demo](https://www.youtube.com/watch?v=J8oJEp4qz5w).

[Additional external initiator reference](https://github.com/smartcontractkit/chainlink/wiki/External-Initiators)

**The External Initiator can only initiate webhook jobs that have been linked to it. Trying to initiate a job that is not linked will give an unauthorised error.**

Deleting an external initiator

To delete an external initiator you must use the remote API. You can do this yourself, like so:

```
text
DELETE http://<your chainlink node>/v2/externalinitiators/<external initiator name>
```

You can alternatively use the chainlink client for convenience:

```
shell
chainlink initiators destroy <NAME>
```

Listing external initiators

To see your installed external initiators:

```
text
GET http://<your chainlink node>/v2/externalinitiators?size=100&page=1
```

Or, using the chainlink client:

```
shell
chainlink initiators list
```

```
external-initiators-introduction.mdx:
```

```

section: nodeOperator
date: Last Modified
title: "Introduction"
whatsnext:
 {
 "Building External Initiators":
"/chainlink-nodes/external-initiators/building-external-initiators",
 "Adding External Initiators to Nodes": "/chainlink-nodes/external-
initiators/external-initiators-in-nodes",
 }

```

```
import { Aside } from "@components"
```

External initiators allow jobs in a node to be initiated depending on some external condition. The ability to create and add external initiators to Chainlink nodes enables blockchain agnostic cross-chain compatibility.

```
<Aside type="note">
 At this time of writing, external initiators do not show up in the bridges
tab. However, they act exactly the same as
 if they did.
</Aside>
```

```
<Aside type="note">
 External initiators are disabled on nodes by default. Set the
FEATUREEXTERNALINITIATORS=true configuration
 variable to enable this feature.
</Aside>
```

Initiator Bridges handle the authentication to and from the External Initiator and where to send the messages. When creating a Bridge two parameters are required:

Only the webhook job type can be initiated using an External Initiator.

The external initiator must be created before the webhook job, and must be referenced by name (whitelisted) in order for that external initiator to be allowed to trigger the given webhook job.

When the External Initiator is created it generates two pairs of credentials: Outgoing and Incoming. The Outgoing Access Key and Secret are used to authenticate messages sent from the Core to the External Initiator. The Incoming Access Key and Secret are used to authenticate messages sent from the External Initiator to the Core.

Then, once you've created the name, bridge, and have the correct access keys for the URL, you can proceed to use the external initiator as if it's a regular initiator in future job specs.

For how to create an external initiator see adding external initiators to nodes.

```
direct-request-existing-job.mdx:
```

```

section: nodeOperator
date: Last Modified
title: "Existing Job Example specs"

```

This is an example v2 (TOML) job spec for returning gas price using etherscan in one Chainlink API Call. Note that the job :

- Uses an external adapter to consume the etherscan API: EtherScan External Adapter. Note that this is done using the bridge task: type="bridge" name="etherscan".
- Calls the fulfillOracleRequest2 function. If you are a node operator, use an Operator contract with this job.

To test this job spec from a smart contract, see this Example.

```

toml
type = "directrequest"
schemaVersion = 1
name = "Etherscan gas price"
maxTaskDuration = "0s"
contractAddress = "YOURORACLECONTRACTADDRESS"
minIncomingConfirmations = 0
observationSource = ""

 decodeLog [type="ethabiDecodeLog"
 abi="OracleRequest(bytes32 indexed specId, address requester,
bytes32 requestId, uint256 payment, address callbackAddr, bytes4
callbackFunctionId, uint256 cancelExpiration, uint256 dataVersion, bytes data)"
 data="$(jobRun.logData)"
 topics="$(jobRun.logTopics)"]

 etherscanFast [type="bridge" name="etherscan" requestData="{\\"data\\":
{\\\"endpoint\\": \\\"gasprice\\\", \\\"speed\\\":\\\"fast\\\" }\\\"}"]
 etherscanAverage [type="bridge" name="etherscan" requestData="{\\"data\\":
{\\\"endpoint\\": \\\"gasprice\\\", \\\"speed\\\":\\\"medium\\\" }\\\"}"]
 etherscanSafe [type="bridge" name="etherscan" requestData="{\\"data\\":
{\\\"endpoint\\": \\\"gasprice\\\", \\\"speed\\\":\\\"safe\\\" }\\\"}"]

 decodeLog -> etherscanFast
 decodeLog -> etherscanAverage
 decodeLog -> etherscanSafe

 gasPriceFast [type=jsonparse path="data,result"]
 gasPriceAverage [type=jsonparse path="data,result"]
 gasPriceSafe [type=jsonparse path="data,result"]

 etherscanFast -> gasPriceFast
 etherscanAverage -> gasPriceAverage
 etherscanSafe -> gasPriceSafe

 gasPriceFast -> encodedData
 gasPriceAverage -> encodedData
 gasPriceSafe -> encodedData

 encodedData [type=ethabiEncode abi="(bytes32 requestId, uint256 fastPrice,
uint256 averagePrice, uint256 safePrice)"
 data="{\\"requestId\\": $(decodeLog.requestId),\\"fastPrice\\": $
(gasPriceFast),\\"averagePrice\\": $(gasPriceAverage),\\"safePrice\\": $
(gasPriceSafe)}"]

 encodeTx [type=ethabiEncode
 abi="fulfillOracleRequest2(bytes32 requestId, uint256 payment,
address callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes

```

```
calldata data)"
 data="{\\"requestId\\": $(decodeLog.requestId), \\"payment\\": $
(decodeLog.payment), \\"callbackAddress\\": $
(decodeLog.callbackAddr), \\"callbackFunctionId\\": $
(decodeLog.callbackFunctionId), \\"expiration\\": $(decodeLog.cancelExpiration),
\\"data\\": $(encodedData)}"]
```

```
 submitTx [type=ethTx to="YOURORACLECONTRACTADDRESS" data="$(encodedTx)"]
 encodedData -> encodedTx -> submitTx
""
```

# direct-request-get-bool.mdx:

```

section: nodeOperator
date: Last Modified
title: "GET > Bool Example Job Spec"

```

This is an example v2 (TOML) job spec for calling any public API, parsing the result then returning a bool in one Chainlink API Call. Note that the job calls the fulfillOracleRequest2 function. If you are a node operator, use an Operator contract with this job.

```
toml
type = "directrequest"
schemaVersion = 1
name = "Get > Bool - (TOML)"
maxTaskDuration = "0s"
contractAddress = "YOURORACLECONTRACTADDRESS"
minIncomingConfirmations = 0
observationSource = ""
 decodeLog [type="ethabiDecodeLog"
 abi="OracleRequest(bytes32 indexed specId, address requester,
bytes32 requestId, uint256 payment, address callbackAddr, bytes4
callbackFunctionId, uint256 cancelExpiration, uint256 dataVersion, bytes data)"
 data="$(jobRun.logData)"
 topics="$(jobRun.logTopics)"]

 decodeBor [type="cborParse" data="$(decodeLog.data)"]
 fetch [type="http" method=GET url="$(decodeBor.get)"]
allowUnrestrictedNetworkAccess="true"]
 parse [type="jsonParse" path="$(decodeBor.path)" data="$(fetch)"]
 encodedData [type="ethabiEncode" abi="(bytes32 requestId, bool value)"
data="{ \\"requestId\\": $(decodeLog.requestId), \\"value\\": $(parse) }"]
 encodedTx [type="ethabiEncode"
 abi="fulfillOracleRequest2(bytes32 requestId, uint256 payment,
address callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes
calldata data)"
 data="{\\"requestId\\": $(decodeLog.requestId), \\"payment\\":
$(decodeLog.payment), \\"callbackAddress\\": $
(decodeLog.callbackAddr), \\"callbackFunctionId\\": $
(decodeLog.callbackFunctionId), \\"expiration\\": $(decodeLog.cancelExpiration),
\\"data\\": $(encodedData)}"
]
 submitTx [type="ethTx" to="YOURORACLECONTRACTADDRESS" data="$(encodedTx)"]

 decodeLog -> decodeBor -> fetch -> parse -> encodedData -> encodedTx ->
submitTx
""
```

```
direct-request-get-bytes.mdx:
```

```

```

```
section: nodeOperator
date: Last Modified
title: "GET > Bytes Example Job Spec"

```

This is an example v2 (TOML) job spec for returning bytes in one Chainlink API Call. Note that the job calls the `fulfillOracleRequest2` function. If you are a node operator, use an Operator contract with this job. To test it from a smart contract, see this [Example](#).

```
toml
type = "directrequest"
schemaVersion = 1
name = "Get > Bytes"
maxTaskDuration = "0s"
contractAddress = "YOURORACLECONTRACTADDRESS"
minIncomingConfirmations = 0
observationSource = ""
 decodeLog [type="ethabidecodeLog"
 abi="OracleRequest(bytes32 indexed specId, address requester,
bytes32 requestId, uint256 payment, address callbackAddr, bytes4
callbackFunctionId, uint256 cancelExpiration, uint256 dataVersion, bytes data)"
 data="$(jobRun.logData)"
 topics="$(jobRun.logTopics)"]

 decodeCbor [type="cborparse" data="$(decodeLog.data)"]
 fetch [type="http" method=GET url="$(decodeCbor.get)"]
allowUnrestrictedNetworkAccess="true"]
 parse [type="jsonparse" path="$(decodeCbor.path)" data="$(fetch)"]
 encodeLarge [type="ethabiencode"
 abi="(bytes32 requestId, bytes data)"
 data="{\\"requestId\\": $(decodeLog.requestId), \\"data\\": $(
(parse))}"
]
 encodeTx [type="ethabiencode"
 abi="fulfillOracleRequest2(bytes32 requestId, uint256 payment,
address callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes
calldata data)"
 data="{\\"requestId\\": $(decodeLog.requestId), \\"payment\\":
$(decodeLog.payment), \\"callbackAddress\\": $(
decodeLog.callbackAddr), \\"callbackFunctionId\\": $(
decodeLog.callbackFunctionId), \\"expiration\\": $(decodeLog.cancelExpiration),
\\"data\\": $(encodeLarge)}"
]

 submitTx [type="ethTx" to="YOURORACLECONTRACTADDRESS" data="$(encodeTx)"]

 decodeLog -> decodeCbor -> fetch -> parse -> encodeLarge -> encodeTx ->
submitTx
""
```

This is an example legacy v1 job spec for returning large responses in one Chainlink API Call.

```
json
{
 "name": "large-word",
```

```

"initiators": [
 {
 "id": 9,
 "jobSpecId": "7a97ff84-93ec-406d-9062-1b2531f9251a",
 "type": "runlog",
 "params": {
 "address": "0xc57B33452b4F7BB189bB5AfaE9cc4aBa1f7a4FD8"
 }
 }
],
"tasks": [
 {
 "jobSpecId": "7a97ff8493ec406d90621b2531f9251a",
 "type": "httpget"
 },
 {
 "jobSpecId": "7a97ff8493ec406d90621b2531f9251a",
 "type": "jsonparse"
 },
 {
 "jobSpecId": "7a97ff8493ec406d90621b2531f9251a",
 "type": "resultcollect"
 },
 {
 "jobSpecId": "7a97ff8493ec406d90621b2531f9251a",
 "type": "ethtx",
 "confirmations": 1,
 "params": {
 "abiEncoding": ["bytes32", "bytes"]
 }
 }
]
}

```

# direct-request-get-int256.mdx:

```

section: nodeOperator
date: Last Modified
title: "GET > Int256 Example Job Spec"

```

This is an example v2 (TOML) job spec for calling any public API, retrieving a number , removing its decimals then returning int256 in one Chainlink API Call. Note that the job calls the fulfillOracleRequest2 function. If you are a node operator, use an Operator contract with this job.

```

toml
type = "directrequest"
schemaVersion = 1
name = "Get > Int256 - (TOML)"
maxTaskDuration = "0s"
contractAddress = "YOURORACLECONTRACTADDRESS"
minIncomingConfirmations = 0
observationSource = ""
 decodeLog [type="ethabidecodeLog"
 abi="OracleRequest(bytes32 indexed specId, address requester,
bytes32 requestId, uint256 payment, address callbackAddr, bytes4
callbackFunctionId, uint256 cancelExpiration, uint256 dataVersion, bytes data)"
 data="$(jobRun.logData)"
 topics="$(jobRun.logTopics)"]

```



```

 decodecbor [type="cborparse" data="$(decodeLog.data)"]
 fetch [type="http" method=GET url="$(decodecbor.get)"]
allowUnrestrictedNetworkAccess="true"]
 parse [type="jsonparse" path="$(decodecbor.path)" data="$(fetch)"]

 multiply [type="multiply" input="$(parse)" times="$(decodecbor.times)"]

 encodedata [type="ethabiencode" abi="(bytes32 requestId, int256 value)"
data="{ \"requestId\": $(decodeLog.requestId), \"value\": $(multiply) }"]
 encodetx [type="ethabiencode"
 abi="fulfillOracleRequest2(bytes32 requestId, uint256 payment,
address callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes
calldata data)"
 data="{\"requestId\": $(decodeLog.requestId), \"payment\":
$(decodeLog.payment), \"callbackAddress\": $
(decodeLog.callbackAddr), \"callbackFunctionId\": $
(decodeLog.callbackFunctionId), \"expiration\": $(decodeLog.cancelExpiration),
\"data\": $(encodedata)}"
]
 submittx [type="ethTx" to="YOURORACLECONTRACTADDRESS" data="$(encodetx)"]

 decodeLog -> decodecbor -> fetch -> parse -> multiply -> encodedata ->
encodetx -> submittx
"""

```

# direct-request-get-string.mdx:

```

section: nodeOperator
date: Last Modified
title: "GET > String Example Job Spec"

```

```
import { CodeSample } from "@components"
```

This is an example v2 (TOML) job spec for returning a string in one Chainlink API Call. Note that the job calls the fulfillOracleRequest2 function. If you are a node operator, use an Operator contract with this job. To test it from a smart contract, see this Example.

```

{/ prettier-ignore /}
<CodeSample src="samples/ChainlinkNodes/jobs/get-string.toml"/>

```

# direct-request-get-uint256.mdx:

```

section: nodeOperator
date: Last Modified
title: "GET > Uint256 Example Job Spec"

```

This is an example v2 (TOML) job spec for calling any public API, retrieving a number, removing its decimals then returning uint256 in one Chainlink API Call. Note that the job calls the fulfillOracleRequest2 function. If you are a node operator, use an Operator contract with this job. To test it from a smart contract, see this Example.

```

toml
type = "directrequest"
schemaVersion = 1
name = "Get > Uint256 - (TOML)"

```

```

maxTaskDuration = "0s"
contractAddress = "YOURORACLECONTRACTADDRESS"
minIncomingConfirmations = 0
observationSource = ""
 decodeLog [type="ethabiDecodeLog"
 abi="OracleRequest(bytes32 indexed specId, address requester,
bytes32 requestId, uint256 payment, address callbackAddr, bytes4
callbackFunctionId, uint256 cancelExpiration, uint256 dataVersion, bytes data)"
 data="$(jobRun.logData)"
 topics="$(jobRun.logTopics)"]

 decodeCbor [type="cborParse" data="$(decodeLog.data)"]
 fetch [type="http" method=GET url="$(decodeCbor.get)"]
allowUnrestrictedNetworkAccess="true"]
 parse [type="jsonParse" path="$(decodeCbor.path)" data="$(fetch)"]

 multiply [type="multiply" input="$(parse)" times="$(decodeCbor.times)"]

 encodedData [type="ethabiEncode" abi="(bytes32 requestId, uint256 value)"
data="{ \"requestId\": $(decodeLog.requestId), \"value\": $(multiply) }"]
 encodeTx [type="ethabiEncode"
 abi="fulfillOracleRequest2(bytes32 requestId, uint256 payment,
address callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes
calldata data)"
 data="{\"requestId\": $(decodeLog.requestId), \"payment\":
$(decodeLog.payment), \"callbackAddress\": $
(decodeLog.callbackAddr), \"callbackFunctionId\": $
(decodeLog.callbackFunctionId), \"expiration\": $(decodeLog.cancelExpiration),
\"data\": $(encodedData)}"
]
 submitTx [type="ethTx" to="YOURORACLECONTRACTADDRESS" data="$(encodeTx)"]

 decodeLog -> decodeCbor -> fetch -> parse -> multiply -> encodedData ->
encodeTx -> submitTx
""

```

# multi-word-job.mdx:

```

section: nodeOperator
date: Last Modified
title: "MultiWord Example Job Spec"

```

This is an example v2 (TOML) job spec for returning multiple responses in 1 Chainlink API Call. Note that the job calls the fulfillOracleRequest2 function. If you are a node operator, use an Operator contract with this job. To test it from a smart contract, see this Example.

```

toml
type = "directrequest"
schemaVersion = 1
name = "multi-word (TOML)"
maxTaskDuration = "0s"
contractAddress = "YOURORACLECONTRACTADDRESS"
minIncomingConfirmations = 0
observationSource = ""
 decodeLog [type="ethabiDecodeLog"
 abi="OracleRequest(bytes32 indexed specId, address requester,
bytes32 requestId, uint256 payment, address callbackAddr, bytes4
callbackFunctionId, uint256 cancelExpiration, uint256 dataVersion, bytes data)"
 data="$(jobRun.logData)"

```

```

 topics="$(jobRun.logTopics)"
 decodecbor [type="cborparse" data="$(decodeLog.data)"]
 decodeLog -> decodecbor
 decodecbor -> btc
 decodecbor -> usd
 decodecbor -> eur
 btc [type="http" method=GET url="$(decodecbor.urlBTC)"
allowunrestrictednetworkaccess="true"]
 btcparse [type="jsonparse" path="$(decodecbor.pathBTC)" data="$(btc)"]
 btcmultiply [type="multiply" input="$(btcparse)", times="100000"]
 btc -> btcparse -> btcmultiply
 usd [type="http" method=GET url="$(decodecbor.urlUSD)"
allowunrestrictednetworkaccess="true"]
 usdparse [type="jsonparse" path="$(decodecbor.pathUSD)" data="$(usd)"]
 usdmultiply [type="multiply" input="$(usdparse)", times="100000"]
 usd -> usdparse -> usdmultiply
 eur [type="http" method=GET url="$(decodecbor.urlEUR)"
allowunrestrictednetworkaccess="true"]
 eurparse [type="jsonparse" path="$(decodecbor.pathEUR)" data="$(eur)"]
 eursmultiply [type="multiply" input="$(eurparse)", times="100000"]
 eur -> eurparse -> eursmultiply
 btcmultiply -> encodemwr
 usdmultiply -> encodemwr
 eursmultiply -> encodemwr
 // MWR API does NOT auto populate the requestId.
 encodemwr [type="ethabiencode"
 abi="(bytes32 requestId, uint256 btc, uint256 usd, uint256
eurs)"
 data="{\\"requestId\\": $(decodeLog.requestId), \\"btc\\": $
(btcmultiply), \\"usd\\": $(usdmultiply), \\"eurs\\": $(eursmultiply)}"
]
 encodetx [type="ethabiencode"
 abi="fulfillOracleRequest2(bytes32 requestId, uint256 payment,
address callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes
calldata data)"
 data="{\\"requestId\\": $(decodeLog.requestId), \\"payment\\":
$(decodeLog.payment), \\"callbackAddress\\": $
(decodeLog.callbackAddr), \\"callbackFunctionId\\": $
(decodeLog.callbackFunctionId), \\"expiration\\": $(decodeLog.cancelExpiration),
\\"data\\": $(encodemwr)}"
]
 submittx [type="ethTx" to="YOURORACLECONTRACTADDRESS" data="$(encodetx)"]
 encodemwr -> encodetx -> submittx
""""

```

# all-jobs.mdx:

```

section: nodeOperator
date: Last Modified
title: "Job Types"

```

```
import { Aside } from "@components"
```

This guide outlines different job types.

Solidity cron jobs

<Aside type="note" title="Chainlink Job Scheduler">

If you need to schedule a contract function call, use the Chainlink Job Scheduler. The Job Scheduler uses the Chainlink

Automation network to execute deployed contract calls on a cron schedule that you define, such as an Ethereum cron job for your dApp.

</Aside>

Executes a job on a schedule. Does not rely on any kind of external trigger.

Spec format

```
toml
type = "cron"
schemaVersion = 1
evmChainID = 1
schedule = "CRONTZ=UTC /20 "
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F01"
observationSource = ""
 fetch [type="http" method=GET url="https://chain.link/ETH-USD"]
 parse [type="jsonparse" path="data,price"]
 multiply [type="multiply" times=100]

 fetch -> parse -> multiply
""
```

Shared fields

See shared fields.

Unique fields

- schedule: the frequency with which the job is to be run. There are two ways to specify this:
  - Traditional UNIX cron format, but with 6 fields, not 5. The extra field allows for "seconds" granularity. Note: you must specify the CRONTZ=... parameter if you use this format.
  - @ shorthand, e.g. @every 1h. This shorthand does not take account of the node's timezone, rather, it simply begins counting down the moment that the job is added to the node (or the node is rebooted). As such, no CRONTZ parameter is needed.

For all supported schedules, please refer to the cron library documentation.

Job type specific pipeline variables

- \$(jobSpec.databaseID): the ID of the job spec in the local database. You shouldn't need this in 99% of cases.
- \$(jobSpec.externalJobID): the globally-unique job ID for this job. Used to coordinate between node operators in certain cases.
- \$(jobSpec.name): the local name of the job.
- \$(jobRun.meta): a map of metadata that can be sent to a bridge, etc.

Direct request jobs

Executes a job upon receipt of an explicit request made by a user. The request is detected via a log emitted by an Oracle or Operator contract. This is similar to the legacy ethlog/runlog style of jobs.

Spec format

```
toml
type = "directrequest"
schemaVersion = 1
evmChainID = 1
name = "example eth request event spec"
```

```
contractAddress = "0x613a38AC1659769640aaE063C651F48E0250454C"
```

Optional fields:

```
requesters = [
 "0xAaAA1F8ee20f5565510b84f9353F1E333e753B7a",
 "0xBbBb70f0E81c6F3430dfDc9fa02fB22bDD818c4E"
]
minContractPaymentLinkJuels = "1000000000000000"
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F02"
minIncomingConfirmations = 10
```

```
observationSource = ""
 ds [type="http" method=GET url="http://example.com"]
 dsparse [type="jsonparse" path="USD"]
 dsmultiply [type="multiply" times=100]

 ds -> dsparse -> dsmultiply
""
```

Shared fields

See shared fields.

Unique fields

- contractAddress: The Oracle or Operator contract to monitor for requests
- requesters: Optional - Allows whitelisting requesters
- minContractPaymentLinkJuels Optional - Allows you to specify a job-specific minimum contract payment
- minIncomingConfirmations Optional - Allows you to specify a job-specific MININCOMINGCONFIRMATIONS value, must be greater than global MININCOMINGCONFIRMATIONS

Job type specific pipeline variables

- \$(jobSpec.databaseID): the ID of the job spec in the local database. You shouldn't need this in 99% of cases.
- \$(jobSpec.externalJobID): the globally-unique job ID for this job. Used to coordinate between node operators in certain cases.
- \$(jobSpec.name): the local name of the job.
- \$(jobRun.meta): a map of metadata that can be sent to a bridge, etc.
- \$(jobRun.logBlockHash): the block hash in which the initiating log was received.
- \$(jobRun.logBlockNumber): the block number in which the initiating log was received.
- \$(jobRun.logTxHash): the transaction hash that generated the initiating log.
- \$(jobRun.logAddress): the address of the contract to which the initiating transaction was sent.
- \$(jobRun.logTopics): the log's topics (indexed fields).
- \$(jobRun.logData): the log's data (non-indexed fields).
- \$(jobRun.blockReceiptsRoot) : the root of the receipts trie of the block (hash).
- \$(jobRun.blockTransactionsRoot) : the root of the transaction trie of the block (hash).
- \$(jobRun.blockStateRoot) : the root of the final state trie of the block (hash).

Examples

```
Get > Uint256 job
```

Let's assume that a user makes a request to an oracle to call a public API, retrieve a number from the response, remove any decimals and return uint256.

- The smart contract example can be found [here](#).
- The job spec example can be found [here](#).

#### Get > Int256 job

Let's assume that a user makes a request to an oracle to call a public API, retrieve a number from the response, remove any decimals and return int256.

- The job spec example can be found [here](#).

#### Get > Bool job

Let's assume that a user makes a request to an oracle to call a public API, retrieve a boolean from the response and return bool.

- The job spec example can be found [here](#).

#### Get > String job

Let's assume that a user makes a request to an oracle and would like to fetch a string from the response.

- The smart contract example can be found [here](#).
- The job spec example can be found [here](#).

#### Get > Bytes job

Let's assume that a user makes a request to an oracle and would like to fetch bytes from the response (meaning a response that contains an arbitrary-length raw byte data).

- The smart contract example can be found [here](#).
- The job spec example can be found [here](#).

#### Multi-Word job

Let's assume that a user makes a request to an oracle and would like to fetch multiple words in one single request.

- The smart contract example can be found [here](#).
- The job spec example can be found [here](#).

#### Existing job

Using an existing Oracle Job makes your smart contract code more succinct. Let's assume that a user makes a request to an oracle that leverages Etherscan External Adapter to retrieve the gas price.

- The smart contract example can be found [here](#).
- The job spec example can be found [here](#).

#### Flux Monitor Jobs

The Flux Monitor job type is for continually-updating data feeds that aggregate responses from multiple oracles. The oracles servicing the feed submit rounds based on several triggers:

- An occasional poll, which must show that there has been sufficient deviation from an offchain data source before a new result is submitted
- New rounds initiated by other oracles on the feeds. If another oracle notices sufficient deviation, all other oracles will submit their current observations as well.
- A heartbeat, which ensures that even if no deviation occurs, we submit a new

result to prove liveness. This can take one of two forms:

- The "idle timer", which begins counting down each time a round is started
- The "drumbeat", which simply ticks at a steady interval, much like a cron job

Spec format

```
toml
type = "fluxmonitor"
schemaVersion = 1
name = "example flux monitor spec"
contractAddress = "0x3cCad4715152693fE3BC4460591e3D3Fbd071b42"
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F03"

threshold = 0.5
absoluteThreshold = 0.0 optional

idleTimerPeriod = "1s"
idleTimerDisabled = false

pollTimerPeriod = "1m"
pollTimerDisabled = false

drumbeatEnabled = true
drumbeatSchedule = "CRONTZ=UTC /20 "

observationSource = ""
 // data source 1
 ds1 [type="http" method=GET url="https://pricesource1.com"
 requestData="{\\"coin\\": \\"ETH\\", \\"market\\": \\"USD\\"}"]
 ds1parse [type="jsonparse" path="data,result"]

 // data source 2
 ds2 [type="http" method=GET url="https://pricesource2.com"
 requestData="{\\"coin\\": \\"ETH\\", \\"market\\": \\"USD\\"}"]
 ds2parse [type="jsonparse" path="data,result"]

 ds1 -> ds1parse -> medianizedanswer
 ds2 -> ds2parse -> medianizedanswer

 medianizedanswer [type=median]
 ""
```

Shared fields

See shared fields.

Unique fields

- contractAddress: the address of the FluxAggregator contract that manages the feed.
- threshold: the percentage threshold of deviation from the previous onchain answer that must be observed before a new set of observations are submitted to the contract.
- absoluteThreshold: the absolute numerical deviation that must be observed from the previous onchain answer before a new set of observations are submitted to the contract. This is primarily useful with data that can legitimately sometimes hit 0, as it's impossible to calculate a percentage deviation from 0.
- idleTimerPeriod: the amount of time (after the start of the last round) after which a new round will be automatically initiated, regardless of any observed offchain deviation.
- idleTimerDisabled: whether the idle timer is used to trigger new rounds.
- drumbeatEnabled: whether the drumbeat is used to trigger new rounds.

- `drumbeatSchedule`: the cron schedule of the drumbeat. This field supports the same syntax as the cron job type (see the cron library documentation for details). CRONTZ is required.
- `pollTimerPeriod`: the frequency with which the offchain data source is checked for deviation against the previously submitted onchain answer.
- `pollTimerDisabled`: whether the occasional deviation check is used to trigger new rounds.
- Notes:
  - For duration parameters, the maximum unit of time is h (hour). Durations of a day or longer must be expressed in hours.
  - If no time unit is provided, the default unit is nanoseconds, which is almost never what you want.

#### Job type specific pipeline variables

- `$(jobSpec.databaseID)`: the ID of the job spec in the local database. You shouldn't need this in 99% of cases.
- `$(jobSpec.externalJobID)`: the globally-unique job ID for this job. Used to coordinate between node operators in certain cases.
- `$(jobSpec.name)`: the local name of the job.
- `$(jobRun.meta)`: a map of metadata that can be sent to a bridge, etc.

#### Keeper jobs

Keeper jobs occasionally poll a smart contract method that expresses whether something in the contract is ready for some onchain action to be performed. When it's ready, the job executes that onchain action.

#### Examples:

- Liquidations
- Rebalancing portfolios
- Rebase token supply adjustments
- Auto-compounding
- Limit orders

#### Spec format

```
toml
type = "keeper"
schemaVersion = 1
evmChainID = 1
name = "example keeper spec"
contractAddress = "0x7b3EC232b08BD7b4b3305BE0C044D907B2DF960B"
fromAddress = "0xa8037A20989AFcBC51798de9762b351D63ff462e"
```

#### Shared fields

See shared fields.

#### Unique fields

- `evmChainID`: The numeric chain ID of the chain on which Chainlink Automation Registry is deployed
- `contractAddress`: The address of the Chainlink Automation Registry contract to poll and update
- `fromAddress`: The Oracle node address from which to send updates
- `externalJobID`: This is an optional field. When omitted it will be generated

#### Offchain reporting jobs

Offchain Reporting (OCR) jobs are used very similarly to Flux Monitor jobs. They update data feeds with aggregated data from many Chainlink oracle nodes.



However, they do this aggregation using a cryptographically-secure offchain protocol that makes it possible for only a single node to submit all answers from all participating nodes during each round (with proofs that the other nodes' answers were legitimately provided by those nodes), which saves a significant amount of gas.

Offchain reporting jobs require the `FEATUREOFFCHAINREPORTING=true` environment variable.

## Bootstrap node

Every OCR cluster requires at least one bootstrap node as a kind of "rallying point" that enables the other nodes to find one another. Bootstrap nodes do not participate in the aggregation protocol and do not submit answers to the feed.

## Spec format

```
toml
type = "offchainreporting"
schemaVersion = 1
evmChainID = 1
contractAddress = "0x27548a32b9aD5D64c5945EaE9Da5337bc3169D15"
p2pBootstrapPeers = [
 "/dns4/chain.link/tcp/1234/p2p/16Uiu2HAm58SP7UL8zsnpeuwHfytLocaqgnyaYKP8wu7qRdri
 xLju",
]
isBootstrapPeer = true
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F05"
```

## Shared fields

See shared fields.

## Unique fields

- `contractAddress`: The address of the `OffchainReportingAggregator` contract.
- `evmChainID`: The chain ID of the EVM chain in which the job will operate.
- `p2pBootstrapPeers`: A list of libp2p dial addresses of the other bootstrap nodes helping oracle nodes find one another on the network. It is used with P2P networking stack V1 as follows:

```
p2pBootstrapPeers = ["/dns4/HOSTNAMEORIP/tcp/PORT/p2p/BOOTSTRAPNODE'SP2PID"]
```

- `p2pv2Boostrappers`: A list of libp2p dial addresses of the other bootstrap nodes helping oracle nodes find one another on the network. It is used with P2P networking stack V2 as follows:

```
p2pv2Boostrappers = ["BOOTSTRAPNODE'SP2PID@HOSTNAMEORIP:PORT"]
```

- `isBootstrapPeer`: This must be set to true.

## Job type specific pipeline variables

- `$(jobSpec.databaseID)`: The ID of the job spec in the local database. You shouldn't need this in 99% of cases.
- `$(jobSpec.externalJobID)`: The globally-unique job ID for this job. Used to coordinate between node operators in certain cases.
- `$(jobSpec.name)`: The local name of the job.
- `$(jobRun.meta)`: A map of metadata that can be sent to a bridge, etc.

## Oracle node

Oracle nodes, on the other hand, are responsible for submitting answers.

Spec format

```
toml
type = "offchainreporting"
schemaVersion = 1
evmChainID = 1
name = "OCR: ETH/USD"
contractAddress = "0x613a38AC1659769640aaE063C651F48E0250454C"
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F06"
p2pPeerID = "12D3KooWApUJaQB2saFjyEUfq6BmysnsSnhLnY5CF9tURYVKgoXK"
p2pBootstrapPeers = [

"/dns4/chain.link/tcp/1234/p2p/16Uiu2HAm58SP7UL8zsnpeuwHfytLocaqgnyaYKP8wu7qRdri
xLju",
]
isBootstrapPeer = false
keyBundleID =
"7f993fb701b3410b1f6e8d4d93a7462754d24609b9b31a4fe64a0cb475a4d934"
monitoringEndpoint = "chain.link:4321"
transmitterAddress = "0xF67D0290337bca0847005C7ffd1BC75BA9AAE6e4"
observationTimeout = "10s"
blockchainTimeout = "20s"
contractConfigTrackerSubscribeInterval = "2m"
contractConfigTrackerPollInterval = "1m"
contractConfigConfirmations = 3
observationSource = ""
 // data source 1
 ds1 [type="bridge" name=ethusd]
 ds1parse [type="jsonparse" path="one,two"]
 ds1multiply [type="multiply" times=100]

 // data source 2
 ds2 [type="http" method=GET url="https://chain.link/ethusd"
 requestData="{\\"hi\\": \\"hello\\"}"]
 ds2parse [type="jsonparse" path="three,four"]
 ds2multiply [type="multiply" times=100]

 ds1 -> ds1parse -> ds1multiply -> answer
 ds2 -> ds2parse -> ds2multiply -> answer

 answer [type=median]
""
```

Shared fields

See shared fields.

Unique fields

- contractAddress: The address of the OffchainReportingAggregator contract.
- evmChainID: The chain ID of the EVM chain in which the job will operate.
- p2pPeerID: The base58-encoded libp2p public key of this node.
- p2pBootstrapPeers: A list of libp2p dial addresses of the other bootstrap nodes helping oracle nodes find one another on the network. It is used with P2P networking stack V1 as follows:  
p2pBootstrapPeers = [ "/dns4/<host name or ip>/tcp/<port>/p2p/<bootstrap node's P2P ID>" ]
- p2pv2Boostrappers: A list of libp2p dial addresses of the other bootstrap nodes helping oracle nodes find one another on the network. It is used with P2P networking stack V2 as follows:

```
p2pv2Boostrappers = ["<bootstrap node's P2P ID>@<host name or ip>:<port>"]
```

- keyBundleID: The hash of the OCR key bundle to be used by this node. The Chainlink node keystore manages these key bundles. Use the node Key Management UI or the chainlink keys ocr sub-commands in the CLI to create and manage key bundles.
- monitoringEndpoint: The URL of the telemetry endpoint to send OCR metrics to.
- transmitterAddress: The Ethereum address from which to send aggregated submissions to the OCR contract.
- observationTimeout: The maximum duration to wait before an offchain request for data is considered to be failed/unfulfillable.
- blockchainTimeout: The maximum duration to wait before an onchain request for data is considered to be failed/unfulfillable.
- contractConfigTrackerSubscribeInterval: The interval at which to retry subscribing to onchain config changes if a subscription has not yet successfully been made.
- contractConfigTrackerPollInterval: The interval at which to proactively poll the onchain config for changes.
- contractConfigConfirmations: The number of blocks to wait after an onchain config change before considering it worthy of acting upon.

Job type specific pipeline variables

- \$(jobSpec.databaseID): The ID of the job spec in the local database. You shouldn't need this in 99% of cases.
- \$(jobSpec.externalJobID): The globally-unique job ID for this job. Used to coordinate between node operators in certain cases.
- \$(jobSpec.name): The local name of the job.
- \$(jobRun.meta): A map of metadata that can be sent to a bridge, etc.

Webhook Jobs

Webhook jobs can be initiated by HTTP request, either by a user or external initiator.

<Aside type="note">

<p>

You must have ExternalInitiatorsEnabled = true in your config to enable these jobs. See{" "

<a href="/chainlink-nodes/v1/node-config#externalinitiatorsenabled">ExternalInitiatorsEnabled</a> in the config reference for details.

</p>

</Aside>

This is an example webhook job:

```
toml
type = "webhook"
schemaVersion = 1
externalInitiators = [
 { name = "my-external-initiator-1", spec = "{\"foo\": 42}" },
 { name = "my-external-initiator-2", spec = "{}" }
]
observationSource = """
 parserequest [type="jsonparse" path="data,result" data="$
(jobRun.requestBody)"]
 multiply [type="multiply" input="$(parserequest)" times="100"]
 sendtobridge [type="bridge" name="mybridge" requestData="{ \\\"result\\\": $
(multiply) }"]

 """
 parserequest -> multiply -> sendtobridge
 """
```

All webhook jobs can have runs triggered by a logged in user.

Webhook jobs may additionally specify zero or more external initiators, which can also trigger runs for this job. The name must exactly match the name of the referred external initiator. The external initiator definition here must contain a spec which defines the JSON payload that will be sent to the External Initiator on job creation if the external initiator has a URL. If you don't care about the spec, you can simply use the empty JSON object.

Unique fields

- externalInitiators - an array of {name, spec} objects, where name is the name registered with the node, and spec is the job spec to be forwarded to the external initiator when it is created.

Shared fields

See shared fields.

Job type specific pipeline variables

- \$(jobSpec.databaseID): the ID of the job spec in the local database. You shouldn't need this in 99% of cases.
- \$(jobSpec.externalJobID): the globally-unique jobID for this job. Used to coordinate between node operators in certain cases.
- \$(jobSpec.name): the local name of the job.
- \$(jobRun.meta): a map of metadata that can be sent to a bridge, etc.
- \$(jobRun.requestBody): the body of the request that initiated the job run.

# all-tasks.mdx:

```

section: nodeOperator
date: Last Modified
title: "Task Types"

```

This guide outlines different task types.

'Any' task

Returns a random value from the set of inputs passed in.

Parameters

None.

Inputs

Can be anything.

Outputs

A randomly-selected value from the set of inputs.

Example

```
toml
fetch1 [type="http" ...]
fetch2 [type="http" ...]
fetch3 [type="http" ...]
```

```
pickany [type="any"]
```

```
fetch1 -> pickany
fetch2 -> pickany
fetch3 -> pickany
```

pickany will return either the result of fetch1, fetch2, or fetch3.

#### Base64 Decode task

Accepts a base64 encoded string and returns decoded bytes.

##### Parameters

- input: a base64 encoded string.

##### Outputs

Decoded bytes.

##### Example

```
toml
mybase64decodetask [type="base64decode" input="SGVsbG8sIHBsYXlnbmQ="]
```

Given the input SGVsbG8sIHBsYXlnbmQ=, the task will return Hello, playground (as ASCII bytes).

#### Base64 Encode task

Encodes bytes/string into a Base64 string.

##### Parameters

- input: Byte array or string to be encoded.

##### Outputs

String with Base64 encoding of input.

##### Example

```
toml
mybase64encodetask [type="base64encode" input="Hello, playground"]
```

Given the input string "Hello, playground", the task will return "SGVsbG8sIHBsYXlnbmQ=".

#### Bridge task

Bridge tasks make HTTP POST requests to pre-configured URLs. Bridges can be configured via the UI or the CLI, and are referred to by a simple user-specified name. This is the way that most jobs interact with External Adapters.

##### Parameters

- name: an arbitrary name given to the bridge by the node operator.
- requestData (optional): a statically-defined payload to be sent to the external adapter.
- cacheTTL (optional): a duration-formatted string indicating the maximum acceptable staleness for cached bridge responses in case of intermittent

failures. This is disabled by default.

- headers (optional): an array of strings. The number of strings must be even.

Example: foo [type="bridge" name="foo" headers=["\\\"X-Header-1\\\", \\\"value1\\\", \\\"X-Header-2\\\", \\\"value2\\\""]]

## Outputs

A string containing the response body.

## Example

```
toml
mybridgetask [type="bridge"
 name="somebridge"
 requestData="{\\\"data\\\":{\\\"foo\\\": $(foo), \\\"bar\\\": $(
(bar))}"
]
```

## CBOR Parse task

CBOR Parse tasks parse a CBOR payload, typically as part of a Direct Request workflow. In Direct Request, a user makes an onchain request using a ChainlinkClient contract, which encodes the request parameters as CBOR. See below for an example.

## Parameters

- data: A byte array containing the CBOR payload.
- mode: An optional parameter that specifies how to parse the incoming CBOR. The default mode is diet, which expects the input to be a map. Set the mode to standard to pass literal values through "as-is". Empty inputs return nil.

## Outputs

A map containing the request parameters. Parameters can be individually accessed using \$(dot.accessors).

## Example

```
toml
// First, we parse the request log and the CBOR payload inside of it
decode_log [type="ethabidecode_log"
 data="$(jobRun.logData)"
 topics="$(jobRun.logTopics)"
 abi="SomeContractEvent(bytes32 requestID, bytes cborPayload)"]

decode_cbor [type="cborparse"
 data="$(decode_log.cborPayload)"]

// Then, we use the decoded request parameters to make an HTTP fetch
fetch [type="http" url="$(decode_cbor.fetchURL)" method=GET]
parse [type="jsonparse" path="$(decode_cbor.jsonPath)" data="$(fetch)"]

// ... etc ...
```

See the Direct Request page for a more comprehensive example.

## Divide task

Divides the provided input by the divisor and returns the result with a number of decimal places defined in the precision value.

## Parameters

- input: The value to be divided
  - number
  - stringified number
  - bytes-ified number
  - \$(variable)
- divisor: The value by which to divide the input
  - number
  - stringified number
  - bytes-ified number
  - \$(variable)
- precision: The number of decimal places to retain in the result
  - number
  - stringified number
  - bytes-ified number
  - \$(variable)

## Outputs

The result of the division.

## Example

```
toml
mydividetask [type="divide"
 input="$(jsonparseresult)"
 divisor="3"
 precision="2"]
```

Given the input 10, this example returns 3.33.

## ETH ABI Decode Log task

Decodes a log emitted by an ETH contract.

## Parameters

- abi: a canonical ETH log event definition. Should be formatted exactly as in Solidity. Each argument must be named. Examples:
  - NewRound(uint256 indexed roundId, address indexed startedBy, uint256 startedAt)
  - AuthorizedSendersChanged(address[] senders)
- data: the ABI-encoded log data. Can be:
  - a byte array
  - a hex-encoded string beginning with 0x
  - ... but generally should just be set to \$(jobRun.logData) (see the Direct Request page)
- topics: the ABI-encoded log topics (i.e., the indexed parameters)
  - an array of bytes32 values
  - an array of hex-encoded bytes32 values beginning with 0x
  - ... but generally should just be set to \$(jobRun.logTopics) (see the Direct Request page)

## Outputs

A map containing the decoded values.

## Example

```
{/ prettier-ignore /}
toml
decode [type="ethabidecode-log"]
```

```

abi="NewRound(uint256 indexed roundId, address indexed startedBy,
uint256 startedAt)"
data="$(jobRun.logData)"
topics="$(jobRun.logTopics)"]

```

This task will return a map with the following schema:

```

{/ prettier-ignore /}
json
{
 "roundId": ..., // a number
 "startedBy": ..., // an address
 "startedAt": ..., // a number
}

```

## ETH ABI Decode task

Decodes a ETH ABI-encoded payload, typically the result of an ETH Call task.

### Parameters

- abi: a canonical ETH ABI argument string. Should be formatted exactly as in Solidity. Each argument must be named. Examples:
  - uint256 foo, bytes32 bar, address[] baz
  - address a, uint80[3][] u, bytes b, bytes32 b32
- data: the ABI-encoded payload to decode. Can be:
  - a byte array
  - a hex-encoded string beginning with 0x

### Outputs

A map containing the decoded values.

### Example

```

{/ prettier-ignore /}
toml
decode [type="ethabidecode"
abi="bytes32 requestID, uint256 price, address[] oracles"
data="$(ethcallresult)"]

```

This task will return a map with the following schema:

```

{/ prettier-ignore /}
json
{
 "requestID": ..., // [32]byte value
 "price": ..., // a number
 "oracles": [
 "0x859AAa51961284C94d970B47E82b8771942F1980",
 "0x51DE85B0cD5B3684865ECfEedfBAF12777cd0Ff8",
 ...
]
}

```

## ETH ABI Encode task

Encodes a bytes payload according to ETH ABI encoding, typically in order to perform an ETH Call or an ETH Tx.



## Parameters

- abi: a canonical ETH ABI argument string. Should be formatted exactly as in Solidity. Each argument must be named. If a method name is provided, the 4-byte method signature is prepended to the result. Examples:
  - uint256 foo, bytes32 bar, address[] baz
  - fulfillRequest(bytes32 requestId, uint256 answer)
- data: a map of the values to be encoded. The task will make a best effort at converting values to the appropriate types.

## Outputs

A byte array.

## Example

```
toml
encode [type="ethabiencode"
 abi="fulfillRequest(bytes32 requestId, uint256 answer)"
 data="{\\"requestID\\": $(foo), \\"answer\\": $(bar)}"
]
```

## ETH Call task

Makes a non-mutating contract call to the specified contract with the specified data payload.

## Parameters

- contract: the address of the contract to call.
- data: the data to attach to the call (including the function selector).
- gas: the amount of gas to attach to the transaction.
- from: The from address with which the call should be made. Defaults to zero address.
- gasPrice: The gasPrice for the call. Defaults to zero.
- gasTipCap: The gasTipCap (EIP-1559) for the call. Defaults to zero.
- gasFeeCap: The gasFeeCap (EIP-1559) for the call. Defaults to zero.
- gasUnlimited: A boolean indicating if unlimited gas should be provided for the call. If set to true, do not pass the gas parameter.
- evmChainID: Set this optional parameter to transmit on the given chain. You must have the chain configured with RPC nodes for this to work. If left blank, it will use the default chain.

## Outputs

An ABI-encoded byte array containing the return value of the contract function.

## Example

```
toml
encodecall [type="ethabiencode"
 abi="checkUpkeep(bytes data)"
 data="{ \\"data\\": $(upkeepdata) }"
]

call [type="ethcall"
 contract="0xa36085F69e2889c224210F603D836748e7dC0088"
 data="$(encodecall)"
 gas="1000"
]

decoderesult [type="ethabidecode"
 abi="bool upkeepNeeded, bytes performData"
 data="$(call)"
]
```

encodecall -> call -> decoderesult

## ETH Tx task

Makes a mutating transaction to the specified contract with the specified data payload. The transaction is guaranteed to succeed eventually.

### Parameters

- from: one or more addresses of the externally-owned account from which to send the transaction. If left blank, it will select a random address on every send for the given chain ID.
- to: the address of the contract to make a transaction to.
- data: the data to attach to the call (including the function selector). Most likely, this will be the output of an ethabiencode task.
- gasLimit: the amount of gas to attach to the transaction.
- txMeta: a map of metadata that is saved into the database for debugging.
- minConfirmations: minimum number of confirmations required before this task will continue. Set to zero to continue immediately. Note that this does not affect transaction inclusion. All transactions will always be included in the chain up to the configured finality depth.
- evmChainID: set this optional parameter to transmit on the given chain. You must have the chain configured with RPC nodes for this to work. If left blank, it will use the default chain.
- failOnRevert: an optional parameter, a boolean, that allows a ChainLink node operator's UI to display and color the status of the task within a job's pipeline depending on a transaction status. default: false.

### Outputs

The hash of the transaction attempt that eventually succeeds (after potentially going through a gas bumping process to ensure confirmation).

### Example

```
toml
encodetx [type="ethabiencode"
 abi="performUpkeep(bytes performData)"
 data="{ \"data\": $(upkeepdata) }"]

submittx [type="ethtx"
 to="0xa36085F69e2889c224210F603D836748e7dC0088"
 data="$(encodetx)"
 failOnRevert="true"]
```

encodetx -> submittx

## Hex Decode task

Accepts a hexadecimal encoded string and returns decoded bytes.

### Parameters

- input: a hexadecimal encoded string, must have prefix 0x.

### Outputs

Decoded bytes.

### Example

```
toml
```

```
myhexdecodetask [type="hexdecode" input="0x12345678"]
```

Given the input 0x12345678, the task will return [0x12, 0x34, 0x56, 0x78].

#### Hex Encode task

Encodes bytes/string/integer into a hexadecimal string.

#### Parameters

- input: Byte array, string or integer to be encoded.

#### Outputs

Hexadecimal string prefixed with "0x" (or empty string if input was empty).

#### Example

```
toml
myhexencodetask [type="hexencode" input="xyz"]
```

Given the input string "xyz", the task will return "0x78797a", which are the ascii values of characters in the string.

#### HTTP task

HTTP tasks make HTTP requests to arbitrary URLs.

#### Parameters

- method: the HTTP method that the request should use.
- url: the URL to make the HTTP request to.
- requestData (optional): a statically-defined payload to be sent to the external adapter.
- allowUnrestrictedNetworkAccess (optional): permits the task to access a URL at localhost, which could present a security risk. Note that Bridge tasks allow this by default.
- headers (optional): an array of strings. The number of strings must be even. Example: foo [type=http headers=["\\\"X-Header-1\\\"", "\\\"value1\\\"", "\\\"X-Header-2\\\"", "\\\"value2\\\""]]

#### Outputs

A string containing the response body.

#### Example

```
toml
myhttptask [type="http"
 method=PUT
 url="http://chain.link"
 requestData="{\\\"foo\\\": $(foo), \\\"bar\\\": $(bar), \\\"jobID\\\":
123}"
 allowUnrestrictedNetworkAccess=true
]
```

#### JSON Parse task

JSON Parse tasks parse a JSON payload and extract a value at a given keypath.

#### Parameters

- data: the JSON string. Can be:
  - string
  - byte array
- path: the keypath to extract. Must be a comma-delimited list of keys, or specify a custom separator alternative.
- separator: (optional) custom path key separator. Defaults to comma (,).
- lax (optional): if false (or omitted), and the keypath doesn't exist, the task will error. If true, the task will return nil to the next task.

#### Outputs

The value at the provided keypath.

#### Example

```
toml
myjsontask [type="jsonparse"
 data="$(httpfetchresult)"
 path="data,0,price"]
```

This task returns 123.45 (float64) when given the following example data value:

```
json
{
 "data": [{ "price": 123.45 }, { "price": 678.9 }]
}
```

#### Length task

Returns the length of a byte array or string.

#### Parameters

- input: Byte array, or string to get the length for.

#### Outputs

The length of the byte array or string.

Note: For strings containing multi-byte unicode characters, the output is the length in bytes and not number of characters.

#### Example

```
toml
mylengthtask [type="length" input="xyz"]
```

Given the input string "xyz", the task will return 3, length of the string.

#### Less Than task

Returns a boolean, result of computing left LESSTHAN right.

#### Parameters

- left: the left hand side of comparison. Possible values:
  - number
  - stringified number
  - bytes-ified number
  - \$(variable)

- right: the right hand side of comparison. Possible values:
  - number
  - stringified number
  - bytes-ified number
  - \$(variable)

#### Outputs

The result of less than comparison.

#### Example

```
toml
mylessthan task [type="lessthan" left="3" right="10"]
```

the task will return true which is the result of 3 LESSTHAN 10

#### Lowercase task

Accepts a string and returns a lowercase string.

#### Parameters

- input: a string.

#### Outputs

Lowercase string.

#### Example

```
toml
mylowercase task [type="lowercase" input="Hello World!"]
```

Given the input Hello World!, the task will return hello world!.

#### Mean task

Accepts multiple numerical inputs and returns the mean (average) of them.

#### Parameters

- values: an array of values to be averaged.
- allowedFaults (optional): the maximum number of input tasks that can error without the Mean task erroring. If not specified, this value defaults to N - 1, where N is the number of inputs.
- precision: the number of decimal places in the result.

#### Outputs

The average of the values in the values array.

#### Example

```
toml
mymeantask [type="mean"
 values=<[$(fetch1), $(fetch2), $(fetch3)]>
 precision=2
 allowedFaults=1]
```

Given the inputs 2, 5, and 20, the task will return 9.

## Median task

Accepts multiple numerical inputs and returns the median of them.

### Parameters

- values: an array of values from which to select a median.
- allowedFaults (optional): the maximum number of input tasks that can error without the Median task erroring. If not specified, this value defaults to N - 1, where N is the number of inputs.

### Outputs

The median of the values in the values array.

### Example

```
toml
mymediantask [type="median"
 values=<[$(fetch1), $(fetch2), $(fetch3)]>
 allowedFaults=1]
```

Given the inputs 2, 5, and 20, the task will return 5.

## Memo task

The memo task returns its value as a result.

### Parameters

- value: value to return. Possible values:
  - number
  - boolean
  - float
  - string
  - array

### Outputs

The value.

### Example

```
toml
memo [type="memo" value="10"]
```

The task will return the value 10

## Merge task

Merge task returns the merged value of two maps.

### Parameters

- left: The left map.
- right: The right map, which overwrites the left side.

### Outputs

Returns the combined map of left and right. If the merged map is invalid, it returns null.

## Example

```
toml
merge [type="merge" left="{\\"foo\\":\\"abc\\", \\"bar\\":\\"123\\"}"
right="{\\"bar\\":\\"xyz\\", \\"biz\\":\\"buzz\\"}"]
```

This example task returns the following map:

```
json
{ "foo": "abc", "bar": "xyz", "biz": "buzz" }
```

## Mode task

Accepts multiple numerical inputs and returns the mode (most common) of them. If more than one value occur the maximum number of times, it returns all of them.

### Parameters

- values: an array of values from which to select a mode.
- allowedFaults (optional): the maximum number of input tasks that can error without the Mode task erroring. If not specified, this value defaults to  $N - 1$ , where  $N$  is the number of inputs.

### Outputs

A map containing two keys:

```
{/ prettier-ignore /}
json
{
 "results": [...], // An array containing all of the values that occurred
the maximum number of times
 "occurrences": ..., // The number of times those values occurred
}
```

## Example

```
{/ prettier-ignore /}
toml
mymodetask [type="mode"
 values=<[$(fetch1), $(fetch2), $(fetch3), $(fetch4), $(fetch5),
$(fetch6), $(fetch7), $(fetch8)]>
 allowedFaults=3]
```

This task can handle arrays with mixed types. For example, given a values array containing both strings and ints like [ 2, 5, 2, "foo", "foo" "bar", "foo", 2 ], the task returns the following JSON:

```
{/ prettier-ignore /}
json
{
 "occurrences": 3,
 "results": [2, "foo"]
}
```

To encode the results array into the ethabiencode task, specify that the data is an array in the abi and point the data parameter to the results from your mode task. Because argument encoding enforces types, all of the values in the array

must be of the same type. As an example, you can encode the results of a mode task with an array of integers:

```
{/ prettier-ignore /}
toml
modetask [type="mode" values=<[1, 2, 2, 3, 1]>]

encodemodetask [type="ethabiencode" abi="(bytes32 requestId, uint64 occurrences,
uint256[] results)" data="{\\"requestId\\": $
(decodeLog.requestId), \\"occurrences\\": $
(modetask.occurrences), \\"results\\": $(modetask.results) }"]
```

In this example, the mode task returns the following result:

```
{/ prettier-ignore /}
json
{
 "occurrences": 2,
 "results": [2, 1]
}
```

The ethabiencode task encodes results as a uint256[] array.

#### Multiply task

Multiplies the provided input and times values.

#### Parameters

- input: the value to be multiplied. Possible values:
  - number
  - stringified number
  - bytes-ified number
  - \$(variable)
- times: the value to multiply the input with.
  - number
  - stringified number
  - bytes-ified number
  - \$(variable)

#### Outputs

The result of the multiplication.

#### Example

```
toml
mymultiplytask [type="multiply" input="$(jsonparseresult)" times=3]
```

Given the input 10, the task will return 30.

#### Sum Task

Accepts multiple numerical inputs and returns the sum of them.

#### Parameters

- values: an array of values to sum.
- allowedFaults (optional): the maximum number of input tasks that can error without the Sum task erroring. If not specified, this value defaults to N - 1, where N is the number of inputs.



## Outputs

The sum of the values in the values array.

## Example

```
toml
mysumtask [type="sum"
 values=<[$(fetch1), $(fetch2), $(fetch3)]>
 allowedFaults=1]
```

Given the inputs 2, 5, and 20, the task will return 27.

## Uppercase task

Accepts a string and returns an uppercase string.

## Parameters

- input: a string.

## Outputs

Uppercase string.

## Example

```
toml
myuppercasetask [type="uppercase" input="Hello World!"]
```

Given the input Hello World!, the task will return HELLO WORLD!.

# jobs.mdx:

```

section: nodeOperator
date: Last Modified
title: "v2 Jobs"

```

```
import { Aside } from "@components"
```

```
<Aside type="note" title="Chainlink v2 jobs">
 This page describes Chainlink v2 jobs. In the Operator UI interface, these are
 called TOML jobs. The v1 jobs are
 removed in Chainlink nodes versioned 1.0.0 and later. To learn how to migrate
 your v1 jobs to v2 jobs, see Migrating
 to v2 Jobs. If you still need to use v1 jobs on older versions of
 Chainlink nodes, see the v1 Jobs documentation.
</Aside>
```

## What is a Job?

Chainlink nodes require jobs to do anything useful. For example, posting asset price data onchain requires jobs. Chainlink nodes support the following job types:

- cron
- directrequest
- fluxmonitor

- keeper
- offchainreporting
- webhook

Jobs are represented by TOML specifications.

Example v2 job spec

The following is an example cron job spec. This is a simple spec that you can add to a node:

```
toml
type = "cron"
schemaVersion = 1
schedule = "CRONTZ=UTC 0 0 1 1 "
Optional externalJobID: Automatically generated if unspecified
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F46"
observationSource = ""
ds [type="http" method=GET url="https://chain.link/ETH-USD"];
dsparse [type="jsonparse" path="data,price"];
dsmultiply [type="multiply" times=100];
ds -> dsparse -> dsmultiply;
""
```

Shared fields

Every job type supported by a node shares the following TOML fields:

- name: The name of the job in the Operator UI
- type: Specifies the v2 job type, which can be one of the following:
  - cron
  - directrequest
  - fluxmonitor
  - keeper
  - offchainreporting
  - webhook
- schemaVersion: Must be present and set to a value of 1. This field will handle progressive iterations of the job spec format gracefully with backwards-compatibility.
- observationSource: The v2 pipeline task DAG, which is specified in DOT syntax. See below for information on writing pipeline DAGs.
- maxTaskDuration: The default maximum duration that any task is allowed to run. If the duration is exceeded, the task is errored. This value can be overridden on a per-task basis using the timeout attribute. See the Shared attributes section for details.
- externalJobID: An optional way for a user to deterministically provide the ID of a job. The ID must be unique. For example, you can specify an externalJobID if you want to run the same directrequest job on two different Chainlink nodes with different bridge names. The spec contents differ slightly, but you can use the same externalJobID on both jobs, specify that in your onchain requests, and both nodes will pick it up. If you do not provide an externalJobID, the node generates the ID for you.
- gasLimit: Optional gas limit for any outgoing transactions spawned by this job. When specified, it overrides ETHGASLIMITDEFAULT env variable.
- forwardingAllowed: Optional. When true, it allows forwarding transactions submitted by the job.

# migration-v1-v2.mdx:

---

section: nodeOperator  
date: Last Modified

```
title: "Migrating to v2 Jobs"
```

```

```

Chainlink nodes with version 1.0.0 and later support v2 jobs in TOML format. Support for v1 jobs in JSON format is removed.

Comparison between v1 and v2 jobs

v1 jobs were intended for extremely targeted use cases, so they opted for simplicity in the job spec over explicitness.

The v2 Job Specs support expanding functionality in Chainlink nodes and prefer explicitness, so they are much more powerful and support advanced capabilities like running tasks in parallel. This change provides the following benefits to node operators:

- Support increased job complexity
- Better performance
- Easier scaling
- Ability to run more offchain computing
- Reliability
- Easier support
- Improved security

DAG dependencies and variables

v2 jobs require the author to specify dependencies using DOT syntax>). If a task needs data produced by another task, this must be specified using DOT.

To facilitate explicitness, v2 jobs require the author to specify inputs to tasks using \$(variable) syntax. For example, if an http task feeds data into a jsonparse task, it must be specified like the following example:

```
toml
fetch [type="http" method=GET url="http://chain.link/pricefeeds/ethusd"]

// This task consumes the output of the 'fetch' task in its 'data' parameter
parse [type="jsonparse" path="data,result" data="$(fetch)"]

// This is the specification of the dependency
fetch -> parse
```

Task names must be defined before their opening [ bracket. In this example, the name of the task is fetch. The output of each task is stored in the variable corresponding to the name of the task. In some cases, tasks return complex values like maps or arrays. By using dot access syntax, you can access the elements of these values. For example:

```
toml
// Assume that this task returns the following object:
// { "ethusd": 123.45, "btcusd": 678.90 }
parse [type="jsonparse" path="data" data="$(fetch)"]

// Now, we want to send the ETH/USD price to one bridge and the BTC/USD price to
another:
submitethusd [type="bridge" name="ethusd" requestData="{ \\"data\\":
{ \\"value\\": $(parse.ethusd) } }"]
submitbtcusd [type="bridge" name="btcusd" requestData="{ \\"data\\":
{ \\"value\\": $(parse.btcusd) } }"]

parse -> submitethusd
parse -> submitbtcusd
```

## Quotes

Some tasks, like the bridge tasks above, require you to specify a JSON object. Because the keys of JSON objects must be enclosed in double quotes, you must use the alternative < angle bracket > quotes. Angle brackets also enable multi-line strings, which can be useful when a JSON object parameter is large:

```
toml
submitbtcusd [type="bridge"
 name="btcusd"
 requestData="{\\"data\\":{\\"value\\": $(foo), \\"price\\": $(
(bar), \\"timestamp\\": $(baz)}}"
]
```

## Misc. notes

- Each job type provides a particular set of variables to its pipeline. See the documentation for each job type to understand which variables are provided.
- Each task type provides a certain kind of output variable to other tasks that consume it. See the documentation for each task type to understand their output types.

---

## Example Migrations

Runlog with ETH ABI encoding

v1 spec

This spec relies on CBOR-encoded onchain values for the httpget URL and jsonparse path.

```
json
{
 "name": "Get > Bytes32",
 "initiators": [
 {
 "type": "runlog",
 "params": {
 "address": "YOURORACLECONTRACTADDRESS"
 }
 }
],
 "tasks": [
 {
 "type": "httpget"
 },
 {
 "type": "jsonparse"
 },
 {
 "type": "ethbytes32"
 },
 {
 "type": "ethtx"
 }
]
}
```

Notes:

- In v1, the job ID is randomly generated at creation time. In v2 jobs, the job ID can be manually specified or the Chainlink node will automatically generate it.
- In v2, the ethbytes32 task and all of the other ABI encoding tasks are now encapsulated in the ethabiencode task with much more flexibility. See the ETH ABI Encode task page to learn more.

Equivalent v2 spec:

```
toml
type = "directrequest"
schemaVersion = 1
name = "Get > Bytes32"
contractAddress = "0x613a38AC1659769640aaE063C651F48E0250454C"
Optional externalJobID: Automatically generated if unspecified
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F47"
observationSource = ""
 decodeLog [type="ethabidecodeLog"
 abi="OracleRequest(bytes32 indexed specId, address requester,
bytes32 requestId, uint256 payment, address callbackAddr, bytes4
callbackFunctionId, uint256 cancelExpiration, uint256 dataVersion, bytes data)"
 data="$(jobRun.logData)"
 topics="$(jobRun.logTopics)"]

 decodeCbor [type="cborparse" data="$(decodeLog.data)"]
 fetch [type="http" method=GET url="$(decodeCbor.url)"]
 parse [type="jsonparse" path="$(decodeCbor.path)" data="$(fetch)"]
 encodedData [type="ethabiencode" abi="(uint256 value)" data="{ \"value\":
$(parse) }"]
 encodeTx [type="ethabiencode"
 abi="fulfillOracleRequest(bytes32 requestId, uint256 payment,
address callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes32
data)"
 data="{\"requestId\": $(decodeLog.requestId), \"payment\":
$(decodeLog.payment), \"callbackAddress\": $
(decodeLog.callbackAddr), \"callbackFunctionId\": $
(decodeLog.callbackFunctionId), \"expiration\": $(decodeLog.cancelExpiration),
\"data\": $(encodedData)}"
]
 submitTx [type="ethTx" to="0x613a38AC1659769640aaE063C651F48E0250454C"
data="$(encodeTx)"]

 decodeLog -> decodeCbor -> fetch -> parse -> encodedData -> encodeTx ->
submitTx
""
```

Simple fetch (runlog)

v1 spec:

```
json
{
 "initiators": [
 {
 "type": "RunLog",
 "params": {
 "address": "0x51DE85B0cD5B3684865ECfEedfBAF12777cd0Ff8"
 }
 }
],
 "tasks": [
 {
```

```

 "type": "HTTPGet",
 "params": {
 "get": "https://bitstamp.net/api/ticker/"
 }
 },
 {
 "type": "JSONParse",
 "params": {
 "path": ["last"]
 }
 },
 {
 "type": "Multiply",
 "params": {
 "times": 100
 }
 },
 {
 "type": "EthUint256"
 },
 {
 "type": "EthTx"
 }
],
"startAt": "2020-02-09T15:13:03Z",
"endAt": null,
"minPayment": "10000000000000000000"
}

```

Equivalent v2 spec:

```

toml
type = "directrequest"
schemaVersion = 1
name = "Get > Bytes32"
contractAddress = "0x613a38AC1659769640aaE063C651F48E0250454C"
Optional externalJobID: Automatically generated if unspecified
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F47"
observationSource = ""
 decodeLog [type="ethabidecodeLog"
 abi="OracleRequest(bytes32 indexed specId, address requester,
bytes32 requestId, uint256 payment, address callbackAddr, bytes4
callbackFunctionId, uint256 cancelExpiration, uint256 dataVersion, bytes data)"
 data="$(jobRun.logData)"
 topics="$(jobRun.logTopics)"]

 fetch [type="http" method=get url="https://bitstamp.net/api/ticker/"]
 parse [type="jsonparse" data="$(fetch)" path="last"]
 multiply [type="multiply" input="$(parse)" times=100]
 encodedata [type="ethabiencode" abi="(uint256 value)" data="{ \"value\": $(
(multiply))"}"]
 encodetx [type="ethabiencode"
 abi="fulfillOracleRequest(bytes32 requestId, uint256 payment,
address callbackAddress, bytes4 callbackFunctionId, uint256 expiration, bytes32
data)"
 data="{\"requestId\": $(decodeLog.requestId), \"payment\": $(
decodeLog.payment), \"callbackAddress\": $(
decodeLog.callbackAddr), \"callbackFunctionId\": $(
decodeLog.callbackFunctionId), \"expiration\": $(decodeLog.cancelExpiration),
\"data\": $(encodedata)}"]
]
 submittx [type="ethTx" to="0x613a38AC1659769640aaE063C651F48E0250454C"
data="$(encodetx)"]

```

```
 decode log -> fetch -> parse -> multiply -> encodedata -> encodetx ->
submittx
"""
```

Cron

v1 spec:

```
json
{
 "initiators": [
 {
 "type": "cron",
 "params": {
 "schedule": "CRONTZ=UTC /20 "
 }
 }
],
 "tasks": [
 {
 "type": "HttpGet",
 "params": {
 "get": "https://example.com/api"
 }
 },
 {
 "type": "JsonParse",
 "params": {
 "path": ["data", "price"]
 }
 },
 {
 "type": "Multiply",
 "params": {
 "times": 100
 }
 },
 {
 "type": "EthUint256"
 },
 {
 "type": "EthTx"
 }
]
}
```

Equivalent v2 spec:

```
toml
type = "cron"
schemaVersion = 1
schedule = "CRONTZ=UTC /20 "
Optional externalJobID: Automatically generated if unspecified
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F46"
observationSource = """
 fetch [type="http" method=GET url="https://example.com/api"]
 parse [type="jsonparse" data="$(fetch)" path="data,price"]
 multiply [type="multiply" input="$(parse)" times=100]
 encodetx [type="ethabiencode"
 abi="submit(uint256 value)"
 data="{ \"value\": $(multiply) }"]
```

```

 submittx [type="eth tx" to="0x859AAa51961284C94d970B47E82b8771942F1980"
data="$(encodetx)"]

```

```

 fetch -> parse -> multiply -> encodetx -> submittx
"""

```

Web (-> Webhook)

v1 spec:

```

json
{
 "initiators": [
 {
 "type": "web"
 }
],
 "tasks": [
 {
 "type": "multiply",
 "params": {
 "times": 100
 }
 },
 {
 "type": "custombridge"
 }
]
}

```

Equivalent v2 spec:

```

toml
type = "webhook"
schemaVersion = 1
Optional externalJobID: Automatically generated if unspecified
externalJobID = "0EEC7E1D-D0D2-476C-A1A8-72DFB6633F46"
observationSource = """
 multiply [type="multiply" input="$(jobRun.requestBody)" times=100]
 sendtobridge [type="bridge" name="custombridge" requestData="{ \"data\": {
\\\"value\\\": $(multiply) }}"]

 multiply -> sendtobridge
"""

```

# tasks.mdx:

```

section: nodeOperator
date: Last Modified
title: "Tasks"

```

```

import { Aside } from "@components"

```

What is a Task?

```

<Aside type="note" title="Tasks">
 Tasks replace the core adapters from v1 jobs.
</Aside>

```



Tasks are a replacement for core adapters that is more flexible. Tasks can be composed in arbitrary order into pipelines. Pipelines consist of one or more threads of execution where tasks are executed in a well-defined order.

You can use Chainlink's built-in tasks, or you can create your own external adapters for tasks which are accessed through a bridge.

### Shared attributes

All tasks share a few common attributes:

`index`: when a task has more than one input (or the pipeline overall needs to support more than one final output), and the ordering of the values matters, the `index` parameter can be used to specify that ordering.

```
toml
data1 [type="http" method=GET url="https://chain.link/ethusd" index=0]
data2 [type="http" method=GET url="https://chain.link/ethdominance" index=1]
multiwordabiencode [type="ethabiencode" method="fulfill(uint256,uint256)"]
```

```
data1 -> multiwordabiencode
data2 -> multiwordabiencode
```

`timeout`: The maximum duration that the task is allowed to run before it is considered to be errored. Overrides the `maxTaskDuration` value in the job spec.

### Writing pipelines

Pipelines are composed of tasks arranged in a DAG (directed acyclic graph). Pipelines are expressed in DOT syntax.

Each node in the graph is a task with a user-specified ID and a set of configuration parameters and attributes:

```
toml
myfetchtask [type="http" method=GET url="https://chain.link/ethusd"]
```

The edges between tasks define how data flows from one task to the next. Some tasks can have multiple inputs, such as `median`. Other tasks are limited to 0 (`http`) or 1 (`jsonparse`).

```
toml
datasource1 [type="http" method=GET url="https://chain.link/ethusd"]
datasource2 [type="http" method=GET url="https://coingecko.com/ethusd"]
medianizedata [type="median"]
submittoea [type="bridge" name="mybridge"]
```

```
datasource1 -> medianizedata
datasource2 -> medianizedata
medianizedata -> submittoea
```

### !DAG Example

```
best-security-practices.mdx:

section: nodeOperator
date: Last Modified
title: "Security and Operation Best Practices"
```

---

The following information provides a set of security and operation best practices that node operators need to use at a minimum to enhance the security and reliability of their infrastructure.

### Restricting access

To run a Chainlink node, the Operator UI port does not need to be open on the internet for it to correctly function. Due to this, we strongly recommend restricting access to all of the services required over the internet.

#### Minimum Requirements:

- SSH (port 22 or changed from the default) is open, and access to the node is granted via SSH tunnelling. This is done typically by adding `-L 6688:localhost:6688` to your SSH command.
- Access to the Ethereum client that the Chainlink node uses is restricted to solely the Chainlink node. This includes ports 8545 and 8546, but excludes 30303 for P2P traffic.

#### Recommended:

- The use of a VPN restricts access to only those who are signed into the VPN in order to access internal resources. For example, this can be achieved by using something like OpenVPN Access Server.
- With the use of the VPN, all traffic between Chainlink nodes and Ethereum clients is routed internally rather than over the internet. For example, all servers are placed in an internal subnet range such as `10.0.0.0/16` and use these IP addresses for communicating.
- Chainlink nodes have the potential to send arbitrary HTTP GET and POST requests, exposing internal network resources. We recommend deploying with a DMZ which has strong outbound network restrictions.

### Failover capabilities

To ensure there is very minimal downtime, failover capabilities are required on both the Chainlink and Ethereum clients so that if any one server fails, the service is still online.

#### Minimum requirements:

- Chainlink nodes are using a PostgreSQL database that are not on the same servers as the Chainlink nodes.
- At least two Chainlink nodes are running at any one time, with both of them pointing to the same database to ensure failover if one fails.

#### Ethereum-specific:

- Ethereum client websocket connectivity is fronted by a load balancer, used by the Chainlink nodes. Here is an example on how to set up a load balancer.
  - If a VPN and internal routing is configured, SSL is not needed but still recommended, as all traffic is purely internal.
  - If both Ethereum and Chainlink nodes are public facing without a VPN, SSL is required to ensure that no communication between both can be intercepted.

### Disaster recovery

Problems occur and when they do, the right processes need to be in-place to ensure that as little downtime as possible occurs. The main impediment to incurring large amounts of downtime in the context of Chainlink node operators is a fully corrupted Ethereum node that requires a re-sync.

Due to the challenge of recovering an Ethereum client, we recommend:

- Daily snapshots of the Ethereum chain on a separate server than what the Chainlink node is connected to.
- An Ethereum client start-up process that pulls down the latest template of the chain and syncs it to the latest height.

With this process in-place, the elapsed time of full disaster is kept to a minimum.

### Active monitoring

To be proactive in detecting any issues before or when they occur, active monitoring needs to be in place. The areas where we recommend to monitor are:

- (Minimum Required) ETH balance of the wallet address assigned to the node.
- Errored job runs.
- Operator UI port to be open and responsive. (Usually: 6688)
- Ethereum http and websocket ports to be open and responsive. (Usually: 8545 & 8546)
- Ethereum client disk, RAM and CPU usage.

Monitoring can be set up from the Docker container's output and fed into most major logging providers. For example, you can use Docker's docs to set up the logging driver for Amazon CloudWatch and Google Cloud Logging. You will want to set the

JSONCONSOLE configuration variable to true so that the output of the container is JSON-formatted for logging.

### Frequent updates

Due to the early nature of the software, it may be required to perform frequent updates to your Chainlink node.

On performing system maintenance to update the Chainlink node, follow this guide.

### Jobs and config

The following are suggestions for job specifications and configuration settings for the node.

#### Job Specifications:

- Include the address of your oracle contract address for all RunLog initiated jobs, as shown in the Fulfilling Requests guide.
- Override the global MININCOMINGCONFIRMATIONS config by setting a confirmations field in jobs which perform offchain payments to allow for greater security by making the node ensure the transaction is still valid after X blocks.

#### Configuring Chainlink Nodes:

- MINIMUMCONTRACTPAYMENTLINKJUELS: ensure your required payment amount is high enough to meet the costs of responding onchain.
- MININCOMINGCONFIRMATIONS: this can be set to 0 for common data request jobs. See the bullet above on setting individual confirmations for specific jobs.
- LOGFILEMAXSIZE: Set this to 0 if you're using external log drivers which parse the output from Docker containers. This will save you disk space.
- JSONCONSOLE: Set to true if you're using external log drivers to parse the output of Docker containers. This will make it easier to parse individual fields of the log and set up alerts.

### Addresses

- Chainlink node address: this is the address used by the Chainlink node to sign

and send responses onchain. This address should not be used by any other service since the Chainlink node keeps track of its nonce locally (instead of polling the network each transaction). This address only needs to be funded with ETH to be able to write back to the blockchain.

- Oracle contract address: this is the address that users will send Chainlink requests to, and which your Chainlink node will fulfill them through this contract. For best practice, it should be owned by a secure address in your control, like a hardware or cold wallet, since it has access to your earned LINK funds. You will not need to fund this contract with anything (LINK or ETH). Instead, users will fund the contract with LINK as they send requests to it, and those funds will become available for withdrawal as your node fulfills the requests.

## Infrastructure as Code (IaC)

Running a Chainlink node works well if you template out your infrastructure using tools like Kubernetes or Terraform. The following repositories can assist you with doing that:

- Pega88's Kubernetes & Terraform setup
- SDL Chainlink Kubernetes Deployment
- LinkPool's Terraform Provider
- Ansible hardened Chainlink
- Terraform module for serverless OCR node on AWS
- Terraform module for serverless adapters on AWS

# connecting-to-a-remote-database.mdx:

```

section: nodeOperator
date: Last Modified
title: "Connecting to a Remote Database"
whatsnext: { "Configuring Chainlink": "/chainlink-nodes/v1/configuration" }

```

This guide show you how to set up a PostgreSQL database and connect your Chainlink node to it. Alternatively, you can follow the guides below:

- Amazon AWS
- Azure
- Docker
- Google Cloud

## Obtain information about your database

In order to connect to a remote database, you must obtain information about the database and the server. Note the following database details so you can use them to configure your Chainlink node later:

- Server hostname or IP
- Port
- Username
- Password
- Database name

The user must be the owner of the desired database. On first run, the migrations will create the tables necessary for the Chainlink node.

## Set Your DATABASEURL environment variable

Below is an example for setting the DATABASEURL environment variable for your Chainlink Node's .env configuration file:

```
text DATABASEURL
DATABASEURL=postgresql://[USERNAME]:[PASSWORD]@[SERVER]:[PORT]/[DATABASE]?
sslmode=[SSLMODE]
```

Change the following placeholders to their real values:

- [USERNAME]: The username for the database owner account.
- [PASSWORD]: The password for the database owner account.
- [SERVER]: The hostname or IP address of the database server.
- [PORT]: The port that the database is listening on. The default port for PostgreSQL is 5432.
- [DATABASE]: The name of the database to use for the Chainlink node.
- [SSLMODE]: If you are testing on a database that does not have SSL enabled, you can specify disable so that you don't need to go through the process of configuring SSL on your database. On a production node, set this value to require or verify-full. This requires an encrypted connection between your Chainlink node and the database. See the PostgreSQL documentation to learn about the available SSL modes.

```
enabling-https-connections.mdx:
```

```

section: nodeOperator
date: Last Modified
title: "Enabling HTTPS Connections"
whatsnext: { "Miscellaneous": "/chainlink-nodes/resources/miscellaneous" }

```

```
import { Aside } from "@components"
import { Tabs } from "@components/Tabs"
```

This guide will walk you through how to generate your own self-signed certificates for use by the Chainlink node. You can also substitute self-signed certificates with certificates of your own, like those created by [Let's Encrypt](https://letsencrypt.org/).

```
<Aside type="tip" title="TLS">
 You will need OpenSSL in order to generate your own self-signed certificates.
</Aside>
```

Create a directory `tls/` within your local Chainlink directory:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 text Sepolia
 mkdir /.chainlink-sepolia/tls

 </Fragment>
 <Fragment slot="panel.2">
 text Mainnet
 mkdir /.chainlink/tls

 </Fragment>
</Tabs>
```

Run this command to create a `server.crt` and `server.key` file in the previously created directory:

```
{/ prettier-ignore /}
```

```

<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
</Fragment slot="panel.1">
shell Sepolia
openssl req -x509 -out /.chainlink-sepolia/tls/server.crt -keyout /.chainlink-sepolia/tls/server.key \
 -newkey rsa:2048 -nodes -sha256 -days 365 \
 -subj '/CN=localhost' -extensions EXT -config <(\
 printf "[dn]\nCN=localhost\n[req]\ndistinguishedname = dn\n[EXT]\nsubjectAltName=DNS:localhost\nkeyUsage=digitalSignature\nextendedKeyUsage=serverAuth")
)
</Fragment>
<Fragment slot="panel.2">
shell Mainnet
openssl req -x509 -out /.chainlink/tls/server.crt -keyout
/.chainlink/tls/server.key \
 -newkey rsa:2048 -nodes -sha256 -days 365 \
 -subj '/CN=localhost' -extensions EXT -config <(\
 printf "[dn]\nCN=localhost\n[req]\ndistinguishedname = dn\n[EXT]\nsubjectAltName=DNS:localhost\nkeyUsage=digitalSignature\nextendedKeyUsage=serverAuth")
)
</Fragment>
</Tabs>

```

Next, add the TLSCERTPATH and TLSKEYPATH environment variables to your .env file.

```

shell Shell
echo "TLSCERTPATH=/chainlink/tls/server.crt
TLSKEYPATH=/chainlink/tls/server.key" >> .env

```

If CHAINLINKTLSPORT=0 is present in your .env file, remove it by running:

```

shell Shell
sed -i '/CHAINLINKTLSPORT=0/d' .env

```

Also remove the line that disables SECURECOOKIES by running:

```

shell Shell
code": "sed -i '/SECURECOOKIES=false/d' .env

```

Finally, update your run command to forward port 6689 to the container instead of 6688:

```

{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 cd /.chainlink-sepolia && docker run -p 6689:6689 -v
 /.chainlink-sepolia:/chainlink -it
 --env-file=.env smartcontract/chainlink local n
 </Fragment>
 <Fragment slot="panel.2">
 shell Mainnet
 cd /.chainlink && docker run -p 6689:6689 -v /.chainlink:/chainlink -it --
 </Fragment>
</Tabs>

```

```
env-file=.env
 smartcontract/chainlink local n
```

```
</Fragment>
</Tabs>
```

Now when running the node, you can access it by navigating to <https://localhost:6689> if running on the same machine or with a ssh tunnel.

```
evm-performance-configuration.mdx:
```

```

section: nodeOperator
date: Last Modified
title: "Optimizing EVM Performance"
whatsnext:
 {
 "Performing System Maintenance": "/chainlink-nodes/resources/performing-
system-maintenance",
 "Security and Operation Best Practices": "/chainlink-nodes/resources/best-
security-practices",
 }
metadata:
 title: "Optimizing EVM Performance"
 description: "Configure your Chainlink and EVM nodes for high throughput and
reliability"

```

```
import { Aside } from "@components"
```

The most basic Chainlink node deployment uses the default configuration on only a single primary node with a websocket URL. This configuration is appropriate for small or simple workloads with only a few jobs that execute infrequently. If you need to run hundreds of jobs and thousands of transactions per hour, your Chainlink and RPC nodes will require a more advanced configuration. This guide explains how to configure Chainlink and your EVM nodes for high reliability and throughput.

```
<Aside type="note" title="Note on Ethereum clients">
```

Ethereum clients have bugs. Much work is done on the Chainlink node software to mitigate bugs in various different RPC implementations. This guide helps you understand how to mitigate and work around these bugs.

```
</Aside>
```

### Using multiple nodes

```
<Aside type="note" title="Use multiple nodes">
```

Providing multiple primary nodes can improve performance and reliability.

```
</Aside>
```

Chainlink node version 1.3.0 and later support configurations with multiple primary nodes and send-only nodes with automatic liveness detection and failover. It is no longer necessary to run a load balancing failover RPC proxy between Chainlink and its EVM RPC nodes.

If you are using a failover proxy transparently for commercial node provider services, it will continue to work properly as long as the RPC you are talking to acts just like a standard RPC node.

You can have as many primary nodes as you want. Requests are evenly distributed across all nodes, so the performance increase will be linear as you add more nodes. If a node fails with no heads for several minutes or a failed liveness

check, it is removed from the live pool and all requests are routed to one of the live nodes. If no live nodes are available, the system attempts to use nodes from the list of failed nodes at random.

You can configure as many send-only nodes as you want. Send-only nodes only broadcast transactions and do not process regular RPC calls. Specifying additional send-only nodes uses a minimum number of RPC calls and can help to include transactions faster. Send-only nodes also act as backup if your primary node starts to blackhole transactions.

<Aside type="note" title="Transaction broadcasts">

Transaction broadcasts are always sent to every primary node and send-only node. It is redundant to specify the same

URL for a send-only node as an existing primary node, and it has no effect.  
</Aside>

Here is an example for how to specify the EVMNODES environment variable:

```
shell
export EVMNODES='
[
 {
 "name": "primary1",
 "evmChainId": "137",
 "wsUrl": "wss://endpoint-1.example.com/ws",
 "httpUrl": "http://endpoint-1.example.com/",
 "sendOnly": false
 },
 {
 "name": "primary2",
 "evmChainId": "137",
 "wsUrl": "ws://endpoint-2.example.com/ws",
 "httpUrl": "http://endpoint-2.example.com/",
 "sendOnly": false
 },
 {
 "name": "primary3",
 "evmChainId": "137",
 "wsUrl": "wss://endpoint-3.example.com/ws",
 "httpUrl": "http://endpoint-3.example.com/",
 "sendOnly": false
 },
 {
 "name": "sendonly1",
 "evmChainId": "137",
 "httpUrl": "http://endpoint-4.example.com/",
 "sendOnly": true
 },
 {
 "name": "sendonly2",
 "evmChainId": "137",
 "httpUrl": "http://endpoint-5.example.com/",
 "sendOnly": true
 },
]
```

Send-only nodes are used for broadcasting transactions only, and must support the following RPC calls:

- ethchainId: Returns the chain ID
- ethsendRawTransaction: Both regular and batched
- web3clientVersion: Can return any arbitrary string



## Automatic load balancing and failover

Chainlink node version 1.3.0 and above has built in failover and load balancing for primary nodes. Chainlink always uses round-robin requests across all primary nodes. Chainlink monitors when nodes go offline and stops routing requests to those nodes. If you don't want to use Chainlink's built-in failover, or you want to use an external proxy instead, you can disable failover completely using the following environment variables:

```
text
NODENONEWHEADSTHRESHOLD=0
NODEPOLLFAILURETHRESHOLD=0
NODEPOLLINTERVAL=0
```

- NODENONEWHEADSTHRESHOLD: Controls how long to wait receiving no new heads before marking a node dead
- NODEPOLLFAILURETHRESHOLD: Controls how many consecutive poll failures will disable a node
- NODEPOLLINTERVAL: Controls how often the node will be polled

By default, these environment variables use the following values:

```
text
NODENONEWHEADSTHRESHOLD="3m"
NODEPOLLFAILURETHRESHOLD="5"
NODEPOLLINTERVAL="10s"
```

## Configuring websocket and HTTP URLs

<Aside type="note" title="Note on URLs">

Ideally, every primary node specifies an HTTP URL in addition to the websocket URL.

</Aside>

It is not recommended to configure primary nodes with only a websocket URL. Routing all traffic over only a websocket can cause problems. As a best practices, every primary node must have both websocket and HTTP URLs specified. This allows Chainlink to route almost all RPC calls over HTTP, which tends to be more robust and reliable. The websocket URL is used only for subscriptions. Both URLs must point to the same node because they are bundled together and have the same liveness state.

If you enabled HTTP URLs on all your primary nodes, you can increase the values for the following environment variables:

- ETHRPCDEFAULTBATCHSIZE
- BLOCKHISTORYESTIMATORBATCHSIZE
- ETHLOGBACKFILLBATCHSIZE

By default, these config variables are set conservatively to avoid overflowing websocket frames. In HTTP mode, there are no such limitations. You might be able to improve performance with increased values similar to the following example:

```
text
ETHRPCDEFAULTBATCHSIZE=1000
BLOCKHISTORYESTIMATORBATCHSIZE=100
ETHLOGBACKFILLBATCHSIZE=1000
```

<Aside type="caution">

<p>Do not modify these values unless all primary nodes are configured with

HTTP URLs.</p>  
</Aside>

## Increasing transaction throughput

By default, Chainlink has conservative limits because it must be compliant with standard out-of-the-box RPC configurations. This limits transaction throughput and the performance of some RPC calls.

Before you make any changes to your Chainlink configuration, you must ensure that all of your primary and send-only nodes are configured to handle the increased throughput.

<Aside type="note" title="Transaction throughput">

The best way to improve transaction throughput is to keep the default configuration and use multiple keys to transmit.

Chainlink supports an arbitrary number of keys for any given chain. By default, tasks will round-robin through keys,

but you can assign them individually to keys as well. Assigning tasks to keys is the preferred way to improve

throughput because increasing the max number of in-flight requests can have complicated effects based on the mempool

configurations of other RPC nodes. If you are unable to distribute transmission load across multiple keys, try the

following options to increase throughput.

</Aside>

## Increase ETHMAXQUEUEDTRANSACTIONS

You can increase ETHMAXQUEUEDTRANSACTIONS if you require high burst throughput. Setting this variable to 0 disables any limit and ensures that no transaction are ever dropped. The default is set automatically based on the chain ID and usually is 250. Overriding this value does not require any RPC changes and only affects the Chainlink side.

This represents the maximum number of unbroadcast transactions per key that are allowed to be enqueued before jobs start failing and refusing to send further transactions. It acts as a "buffer" for transactions waiting to be sent. If the buffer is exceeded, transactions will be permanently dropped.

Do not set ETHMAXQUEUEDTRANSACTIONS too high. It acts as a sanity limit and the queue can grow unbounded if you are trying to send transactions consistently faster than they can be confirmed. If you have an issue that must be recovered later, you will have to churn through all the enqueued transactions. As a best practice, set ETHMAXQUEUEDTRANSACTIONS to the minimum possible value that supports your burst requirements or represents the maximum number of transactions that could be sent in a given 15 minute window.

ETHMAXQUEUEDTRANSACTIONS=10000 might be an example where very high burst throughput is needed.

## Increase ETHMAXINFLIGHTTRANSACTIONS

ETHMAXINFLIGHTTRANSACTIONS is another variable that you can increase if you require higher constant transaction throughput. Setting this variable to 0 disables any kind of limit. The default value is 16.

ETHMAXINFLIGHTTRANSACTIONS controls how many transactions are allowed to be broadcast but unconfirmed at any one time. This is a form of transaction throttling.

The default is set conservatively at 16 because this is a pessimistic minimum that go-ethereum will hold without evicting local transactions. If your node is falling behind and not able to get transactions in as fast as they are created,

you can increase this setting.

<Aside type="caution">

If you increase ETHMAXINFLIGHTTRANSACTIONS you must make sure that your ETH node is configured properly

otherwise you can get nonce-gapped and your node will get stuck.

</Aside>

## Optimizing RPC nodes

You can also improve transaction throughput by optimizing RPC nodes. Configure your RPC node to never evict local transactions. For example, you can use the following example configurations:

text Go-Ethereum

[Eth.TxPool]

Locals = ["0xYourNodeAddress1", "0xYourNodeAddress2"] Add your node addresses here

NoLocals = false Disabled by default but might as well make sure

Journal = "transactions.rlp" Make sure you set a journal file

Rejournal = 3600000000000 Default 1h, it might make sense to reduce this to e.g. 5m

PriceBump = 10 Must be set less than or equal to Chainlink's ETHGASBUMPPERCENT

AccountSlots = 16 Highly recommended to increase this, must be greater than or equal to Chainlink's ETHMAXINFLIGHTTRANSACTIONS setting

GlobalSlots = 4096 Increase this as necessary

AccountQueue = 64 Increase this as necessary

GlobalQueue = 1024 Increase this as necessary

Lifetime = 10800000000000 Default 3h, this is probably ok, you might even consider reducing it

If you are using another RPC node, such as Besu or Nethermind, you must look at the documentation for that node to ensure that it will keep at least as many transactions in the mempool for the Chainlink node keys as you have set in ETHMAXINFLIGHTTRANSACTIONS.

The recommended way to scale is to use more keys rather than increasing throughput for one key.

## Remove rejections on expensive transactions

By default, go-ethereum rejects transactions that exceed its built-in RPC gas or txfee caps. Chainlink nodes fatally error transactions if this happens. If you ever exceed the caps, your node will miss transactions.

Disable the default RPC gas and txfee caps on your ETH node in the config using the TOML snippet shown below, or by running go-ethereum with the command line arguments: `--rpc.gascap=0 --rpc.txfeecap=0`.

text

[Eth]

RPCGasCap = 0

RPCTxFeeCap = 0.0

## Arbitrum differences

Arbitrum Nitro runs a fork of go-ethereum internally, but the original flags are not valid. These modified flags are equivalent:

`--node.rpc.gas-cap 0 --node.rpc.tx-fee-cap 0`

## Adjusting minimum outgoing confirmations for high throughput jobs

eth tx tasks have a minConfirmations label that can be adjusted. You can get a minor performance boost if you set this label to 0. Use this if you do not need to wait for confirmations on your eth tx tasks. For example, if you don't need the receipt or don't care about failing the task if the transaction reverts onchain, you can set minConfirmations to 0.

Set the task label similarly to the following example:

```
foo [type=eth tx minConfirmations=0 ...]
```

Note that this only affects the presentation of jobs, and whether they are marked as errored or not. It has no effect on inclusion of the transaction, which is handled with separate logic.

<Aside type="caution">

Do not confuse minConfirmations set on the task with transaction inclusion. The transaction manager always attempts

to get every transaction mined up to EVMFinalityDepth. minConfirmations on the task is a task-specific view of

when the transaction that can be considered final, which might be fewer blocks than EVMFinalityDepth.

</Aside>

Increase ORMMAXOPENCONNS and ORMMAXIDLECONNS

Chainlink can be configured to allow more concurrent database connections than the default. This might improve performance, but be careful not to exceed postgres connection limits. These variables have the following default values:

text

ORMMAXOPENCONNS=20

ORMMAXIDLECONNS=10

You might increase these values to ORMMAXOPENCONNS=50 and ORMMAXIDLECONNS=25 if you have a large and powerful database server with high connection count headroom.

```
miscellaneous.mdx:
```

```

```

```
section: nodeOperator
```

```
date: Last Modified
```

```
title: "Miscellaneous"
```

```
whatsnext: { "Security and Operation Best Practices":
 "/chainlink-nodes/resources/best-security-practices" }
```

```

```

```
import { Aside } from "@components"
```

```
import { Tabs } from "@components/Tabs"
```

Execute Commands Running Docker

In order to interact with the node's CLI commands, you need to be authenticated. This means that you need to access a shell within the Chainlink node's running container first. You can obtain the running container's NAME by running:

```
shell
```

```
docker ps
```

The output will look similar to:



This method is the preferred way to interact with your node wallet. Using other methods to manually interact with the node wallet can cause nonce issues.

Change your API password

<Aside type="note" title="Note for Docker">

If using Docker, you will first need to follow the [Execute Commands Running Docker](#)

guide to enter the running container.

</Aside>

In order to change your password, you first need to log into the CLI by running:

```
shell
chainlink admin login
```

Use your API email and old password in order to authenticate.

Then run the following in order to update the password:

```
shell
chainlink admin chpass
```

It will ask for your old password first, then ask for the new password and a confirmation.

Once complete, you should see a message "Password updated."

Multi-user and Role Based Access Control (RBAC)

See the [Roles and Access Control](#) page.

Key management

In this section, ensure you log into the CLI by executing the following command:

```
shell
chainlink admin login
```

Authenticate using your API email and password.

List ETH keys

To list available Ethereum accounts along with their ETH & LINK balances, nonces, and other metadata, execute the following command:

```
shell
chainlink keys eth list
```

Example:

```
text
ðŸ”” ‘ ETH keys
```

```

Address: 0x2d4f5FBD00E5A4fD53D162cE7EDFdb5b7664C542
EVM Chain ID: 11155111
Next Nonce: 0
ETH: 0.00000000000000000000
LINK: 0
```

```
Disabled: false
Created: 2023-04-26 08:12:51.340348 +0000 UTC
Updated: 2023-04-26 08:12:51.340348 +0000 UTC
```

### Create a new ETH Key

To create a key in the node's keystore alongside the existing keys, run the following command:

```
shell
chainlink keys eth create
```

Example:

```
text
ETH key created.
```

```
ðŸ”” · New key
```

```

Address: 0xd31961E1f62A2FaB824AC3C1A7a332daF8B11eE0
EVM Chain ID: 11155111
Next Nonce: 0
ETH: 0.000000000000000000
LINK: 0
Disabled: false
Created: 2023-04-26 08:28:36.52974 +0000 UTC
Updated: 2023-04-26 08:28:36.52974 +0000 UTC
Max Gas Price Wei:
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

### Export an ETH key

To export an Ethereum key to a JSON file, run the following command:

```
shell
chainlink keys eth export [address] [command options]
```

where:

- address: The EVM account address for which you want to export the private key.
- Options:
  - --newpassword FILE, -p FILE FILE: A file containing the password to encrypt the key.
  - --output value, -o value: The path where the JSON file will be saved.

Example:

```
text
chainlink keys eth export 0xd31961E1f62A2FaB824AC3C1A7a332daF8B11eE0 --
newpassword .chainlink/pass --output privatekey.json
```

```
ðŸ”” · Exported ETH key 0xd31961E1f62A2FaB824AC3C1A7a332daF8B11eE0 to
privatekey.json
```

### Delete an ETH key

To remove an Ethereum key, run the following command:

```
shell
chainlink keys eth delete [address] [command options]
```

where:

- address: The EVM account address that you want to remove.
- Options:
  - --yes, -y: Skip the confirmation prompt.

Example:

```
text
$ chainlink keys eth delete 0xd31961E1f62A2FaB824AC3C1A7a332daF8B11eE0 --yes
Deleted ETH key: 0xd31961E1f62A2FaB824AC3C1A7a332daF8B11eE0
```

Import an ETH key

To import an Ethereum key from a JSON file, run the following command:

```
shell
chainlink keys eth import [JSON file] [command options]
```

where:

- JSON file: The path where the JSON file containing the ETH key is saved.
- Options:
  - --oldpassword FILE, -p FILE: FILE containing the password used to encrypt the key in the JSON file.
  - --evmChainID value: Chain ID for the key. If left blank, default chain will be used.

Example:

```
text
$ chainlink keys eth import privatekey.json --oldpassword .chainlink/pass
ðŸ”” Imported ETH key
```

```

Address: 0xd31961E1f62A2FaB824AC3C1A7a332daF8B11eE0
EVM Chain ID: 11155111
Next Nonce: 0
ETH: 0.00000000000000000000
LINK: 0
Disabled: false
Created: 2023-04-26 08:51:02.04186 +0000 UTC
Updated: 2023-04-26 08:51:02.04186 +0000 UTC
Max Gas Price Wei: <nil>
```

Full example in detached mode

```
shell
cd /.chainlink-sepolia && docker run --restart=always -p 6688:6688 -d --name
sepolia-primary -v /.chainlink-sepolia:/chainlink -it --env-file=.env
smartcontract/chainlink:1.0.0 node start -p /chainlink/.password
```



```
performing-system-maintenance.mdx:
```

```

```

```
section: nodeOperator
date: Last Modified
title: "Performing System Maintenance"
whatsnext: { "Connecting to a Remote Database":
"/chainlink-nodes/resources/connecting-to-a-remote-database" }

```

```
import { Aside } from "@components"
import { Tabs } from "@components/Tabs"
```

You might occasionally need to restart the system that the Chainlink node runs on. To restart without any downtime for completing requests, perform the upgrade as a series of steps that passes database access to a new instance while the first instance is down.

Maintenance and image update example

```
<Aside type="note" title="Docker">
 This example uses Docker to run the Chainlink node, see the Running a
Chainlink
 Node page for instructions on how to set it up.
</Aside>
```

First, find the most recent Chainlink image on Docker Hub and pull that Docker image. For version 1.11.0:

```
shell
docker pull smartcontract/chainlink:1.11.0
```

Then, check what port the existing container is running on:

```
shell
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND
2d203191c1d6	smartcontract/chainlink:latest	./chainlink-launcher
26 seconds ago	Up 25 seconds	0.0.0.0:6688->6688/tcp jovialshirley

Look under the PORTS label to see the ports in use by the running container, in this case, the local port 6688 has been mapped to the application's port 6688, as identified by the -> arrow. Since we can't use the same local port number twice, we'll need to run the second instance with a different one.

Now start the second instance of the node. The local port option has been modified so that both containers run simultaneously.

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 cd /.chainlink-sepolia && docker run -p 6687:6688 -v
```

```

/.chainlink-sepolia:/chainlink -it
--env-file=.env smartcontract/chainlink local n

</Fragment>
<Fragment slot="panel.2">
 shell Mainnet
 cd /.chainlink && docker run -p 6687:6688 -v /.chainlink:/chainlink -it --
env-file=.env
 smartcontract/chainlink local n

</Fragment>
</Tabs>

```

The log messages on the second node instance inform you that it is waiting for the database lock.

Now you can shut down the first node instance. We'll use the name given earlier and kill the container. Note that your container name will likely be different.

```

shell
docker kill jovialshirley

```

The output returns the name "jovialshirley" (or what your container's name was) and if you look at the log of your second container, you'll notice that it has taken over.

At this point, you're now running the latest image on your secondary container. If you have any system maintenance to perform on your primary machine, you can do so now.

Next, run the container again with the local port 6688 in order to go back to normal operations.

```

{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 cd /.chainlink-sepolia && docker run -p 6688:6688 -v
/.chainlink-sepolia:/chainlink -it
 --env-file=.env smartcontract/chainlink local n

 </Fragment>
 <Fragment slot="panel.2">
 shell Mainnet
 cd /.chainlink && docker run -p 6688:6688 -v /.chainlink:/chainlink -it --
env-file=.env
 smartcontract/chainlink local n

 </Fragment>
</Tabs>

```

When the log messages on the first node indicate that it is waiting for the database lock, shut down the second instance of the node. The original instance automatically obtains a lock and resumes normal operation.

Failover node example

```

<Aside type="note" title="Docker">
 This example uses Docker to run the Chainlink node, see the Running a
Chainlink
 Node page for instructions on how to set it up.

```

</Aside>

You might want to run multiple instances of the Chainlink node on the same machine. If one instance goes down, the second instance can automatically pick up requests. Building off the concepts in the previous example, use Docker to have primary and a secondary containers referencing the same database URL.

Use the default DATABASELOCKINGMODE=advisorylock setting unless you want to test the lease or dual settings. See the docs for more information about this configuration variable.

Run the Chainlink node with a name option specified:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 cd /.chainlink-sepolia && docker run --name chainlink -p 6688:6688 -v
 /.chainlink-sepolia:/chainlink -it --env-file=.env smartcontract/chainlink
 local n

 </Fragment>
 <Fragment slot="panel.2">
 shell Mainnet
 cd /.chainlink && docker run --name chainlink -p 6688:6688 -v
 /.chainlink:/chainlink -it
 --env-file=.env smartcontract/chainlink local n

 </Fragment>
</Tabs>
```

You will now notice that you no longer receive a randomly generated name from Docker:

```
shell
docker ps
```

Output (truncated):

```
... NAMES
... chainlink
```

This will remain your primary Chainlink container, and should always use port 6688 (unless configured otherwise). For the secondary instance, you will run the container in the same way, but with a different name and a different local port:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 cd /.chainlink-sepolia && docker run --name secondary -p 6687:6688 -v
 /.chainlink-sepolia:/chainlink -it --env-file=.env smartcontract/chainlink
 local n

 </Fragment>
 <Fragment slot="panel.2">
 shell Mainnet
```

```
cd /.chainlink && docker run --name secondary -p 6687:6688 -v
/./chainlink:/chainlink -it
--env-file=.env smartcontract/chainlink local n
```

</Fragment>  
</Tabs>

Notice the `--name secondary` was used for this container and the local port is 6687. Be sure to add this port to your SSH tunnel as well so that you can access the secondary node's GUI if it has become active (it will not function until the primary container goes down).

Running `docker ps` now reveals two named containers running (output truncated):

```
... NAMES
... secondary
... chainlink
```

If your primary container goes down, the secondary one automatically takes over. To start the primary container again, simply run:

```
shell
docker start -i chainlink
```

This starts the container, but the secondary node still has a lock on the database. To give the primary container access, you can restart the secondary container:

```
shell
docker restart secondary -t 0
```

The primary container takes control of the database and resumes operation. You can attach to the secondary container using `docker attach`:

```
shell
docker attach secondary
```

However, it does not produce any output while waiting for a lock on the database.

Congratulations! You now have a redundant setup of Chainlink nodes in case the primary container goes down. Get comfortable with the process by passing control of the database back and forth between the chainlink and secondary containers.

```
requirements.mdx:
```

```

section: nodeOperator
date: Last Modified
title: "Requirements"

```

## Hardware

The requirements for running a Chainlink node scale with the as the number of jobs that your node services. CPUs with the x86 architecture is recommended for production environments, but you can use Apple M1 systems for development if you run the Chainlink node in Docker.

- Minimum: At least 2 CPU cores and 4 GB of RAM will allow you to get a node running for testing and basic development.
- Recommended: For nodes in a production environment with over 100 jobs, you will need at least 4 CPU cores and 8GB of RAM.

If you run your PostgreSQL database locally, you will need additional hardware. To support more than 100 jobs, your database server will need at least 4 cores, 16 GB of RAM, and 100 GB of storage.

If you run your node on AWS or another cloud platform, use a VM instance type with dedicated core time. Burstable Performance Instances and VM instances with shared cores often have a limited number of CPU credits, which do not perform well for Chainlink nodes that require consistent performance.

## Software

Chainlink nodes have the following software dependencies:

- Operating System: Linux, MacOS, or the WSL (Windows Subsystem for Linux)
  - For production environments, Linux is recommended.
- Docker: Although it is possible to build Chainlink nodes from source, the best practice is to use the Chainlink Docker Images without `-root`.
- PostgreSQL versions  $\geq 12$  (Version 12 and later).
  - If you use a database as a service, your database host must provide access to logs.
  - If you run the database on a separate system, secure the TCP/IP connection with SSL.

## Blockchain connectivity

Chainlink nodes require a fully-synced network client so that they can run onchain transactions and interact with deployed contracts. For Ethereum, see the list of supported clients. Other L1s, L2s, and side-chains use different clients. See your network's documentation to learn how to run a client for your specific network.

The client must meet the following requirements:

- You can use a provider like Alchemy or Infura, but running your own client can provide lower latency and greater decentralization.
- Run your Chainlink nodes on their own separate VM or system. Hardware and storage requirements for these clients will change over time, so you will likely need to scale their capacity separately from the system where you run your Chainlink nodes.
- The client must provide both HTTP and WebSocket connections secured with SSL. Most providers give you `https://` and `wss://` connections by default. If you run your own client, you must create a reverse proxy for your client using a web server like Nginx. The web server handles the SSL encryption and forwards the connection to your client.

See [Running Ethereum Clients](#) for more details.

```
run-an-ethereum-client.mdx:
```

```

```

```
section: nodeOperator
```

```
date: Last Modified
```

```
title: "Run an Ethereum Client"
```

```
whatsnext:
```

```
{
```

```
 "Running a Chainlink Node": "/chainlink-nodes/v1/running-a-chainlink-node",
 "Optimizing Performance": "/chainlink-nodes/resources/evm-performance-
```

```
configuration",
 }

```

```
import { Aside } from "@components"
import { Tabs } from "@components/Tabs"
```

Chainlink nodes must be able to connect to an Ethereum client with an active websocket connection. This is accomplished by running both an execution client and a consensus client. You can run these clients yourself, but running Ethereum clients requires significant storage and network resources. Optionally, you can use External Services that manage these clients for you.

<Aside type="note" title="Enable the Websockets API">

If you run these clients yourself, you must enable the websockets API. The websockets API is required for the

Chainlink node to communicate with the Ethereum blockchain.

</Aside>

## Geth

You can use the Geth client for the Sepolia testnet and the Ethereum Mainnet. See the Geth Documentation for a list of supported networks.

Download the latest version:

```
shell
docker pull ethereum/client-go:latest
```

Create a local directory to persist the data:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 mkdir /.geth-sepolia

 </Fragment>
 <Fragment slot="panel.2">
 shell Mainnet
 mkdir /.geth

 </Fragment>
</Tabs>
```

Run the container:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 docker run --name eth -p 8546:8546 -v /.geth-sepolia:/geth -it \
 ethereum/client-go --sepolia --ws --ipcdisable \
 --ws.addr 0.0.0.0 --ws.origins="" --datadir /geth

 </Fragment>
 <Fragment slot="panel.2">
 shell Mainnet
 docker run --name eth -p 8546:8546 -v /.geth:/geth -it \
```

```
ethereum/client-go --ws --ipcdisable \
--ws.addr 0.0.0.0 --ws.origins="" --datadir /geth
```

```
</Fragment>
</Tabs>
```

Once the Ethereum client is running, you can use Ctrl + P, Ctrl + Q to detach from the container without stopping it. You will need to leave the container running for the Chainlink node to connect to it.

If the container was stopped and you need to run it again, you can simply use the following command:

```
shell
docker start -i eth
```

Follow Geth's instructions for Connecting to Consensus Clients. This will require some additional configuration settings for the Docker command that runs Geth.

Return to Running a Chainlink Node.

## Nethermind

You can use the Nethermind client for the Ethereum Mainnet. See the Nethermind supported network configurations page for a list of supported networks.

Download the latest version:

```
shell
docker pull nethermind/nethermind:latest
```

Create a local directory to persist the data:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 mkdir /.nethermind-sepolia

 </Fragment>
 <Fragment slot="panel.2">
 shell Mainnet
 mkdir /.nethermind

 </Fragment>
</Tabs>
```

Run the container:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 docker run --name eth -p 8545:8545 \
 -v /.nethermind-sepolia:/nethermind/data \
 -it nethermind/nethermind:latest --config sepolia \
 --Init.WebSocketsEnabled true --JsonRpc.Enabled true \
 </Fragment>
</Tabs>
```

```
--JsonRpc.Host 0.0.0.0 --NoCategory.CorsOrigins \
--datadir data
```

</Fragment>

<Fragment slot="panel.2">

```
shell Mainnet
```

```
docker run --name eth -p 8545:8545 \
-v /.nethermind/./nethermind/data \
-it nethermind/nethermind:latest --Sync.FastSync true \
--Init.WebSocketsEnabled true --JsonRpc.Enabled true \
--JsonRpc.Host 0.0.0.0 --NoCategory.CorsOrigins \
--datadir data
```

</Fragment>

</Tabs>

After the Ethereum client is running, you can use Ctrl + P, Ctrl + Q to detach from the container without stopping it. You will need to leave the container running for the Chainlink node to connect to it.

If the container was stopped and you need to run it again, use the following command to start it:

```
shell
```

```
docker start -i eth
```

Follow Nethermind's instructions for Installing and configuring the Consensus Client. This will require some additional configuration settings for the Docker command that runs Nethermind.

Return to Running a Chainlink Node.

## External Services

The following services offer Ethereum clients with websockets connectivity known to work with the Chainlink node.

### Alchemy

Example connection setting:

```
{/ prettier-ignore /}
```

<Tabs client:visible>

<Fragment slot="tab.1">Sepolia</Fragment>

<Fragment slot="tab.2">Mainnet</Fragment>

<Fragment slot="panel.1">

```
text Sepolia
```

```
ETHURL=wss://eth-sepolia.alchemyapi.io/v2/YOURPROJECTID
```

</Fragment>

<Fragment slot="panel.2">

```
text Mainnet
```

```
ETHURL=wss://eth-mainnet.alchemyapi.io/v2/YOURPROJECTID
```

</Fragment>

</Tabs>

### Chainstack

Example connection setting:

```
text Mainnet
```

```
ETHURL=wss://user-name:pass-word-pass-word-pass-word@ws-nd-123-456-
```



789.p2pify.com

## Fiews

Example connection setting:

```
text Mainnet
ETHURL=wss://cl-main.fiews.io/v2/YOURAPIKEY
```

## GetBlock

Example connection setting:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 text Sepolia
 ETHURL=wss://eth.getblock.io/sepolia/?apikey=YOURAPIKEY

 </Fragment>
 <Fragment slot="panel.2">
 text Mainnet
 ETHURL=wss://eth.getblock.io/mainnet/?apikey=YOURAPIKEY

 </Fragment>
</Tabs>
```

## Infura

Example connection setting. Replace YOURPROJECTID with the ID Infura provides you on your project settings page.

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 text Sepolia
 ETHURL=wss://sepolia.infura.io/ws/v3/YOURPROJECTID

 </Fragment>
 <Fragment slot="panel.2">
 text Mainnet
 ETHURL=wss://mainnet.infura.io/ws/v3/YOURPROJECTID

 </Fragment>
</Tabs>
```

## LinkPool

Example connection setting:

```
text Mainnet
ETHURL=wss://main-rpc.linkpool.io/ws
```

## QuikNode

Example connection setting:

```

{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="tab.2">Mainnet</Fragment>
 <Fragment slot="panel.1">
 text Sepolia
 ETHURL=wss://your-node-name.sepolia.quiknode.pro/security-hash/

 </Fragment>
 <Fragment slot="panel.2">
 text Mainnet
 ETHURL=wss://your-node-name.quiknode.pro/security-hash/

 </Fragment>
</Tabs>

```

## Configuring your ETH node

```

<Aside type="caution" title="RPC gas/txfee caps">
 By default, go-ethereum rejects transactions that exceed the built-in RPC
 gas/txfee caps. The node will fatally error
 transactions if this happens. If you ever exceed the caps, the node will miss
 transactions.
</Aside>

```

At a minimum, disable the default RPC gas and txfee caps on your ETH node. This can be done in the TOML file as seen below, or by running go-ethereum with the command line arguments: `--rpc.gas cap=0 --rpc.txfee cap=0`.

To learn more about configuring ETH nodes, see the configuration page.

## Additional Tools

- Chainlink ETH Failover Proxy

# fulfilling-requests.mdx:

```

section: nodeOperator
date: Last Modified
title: "Fulfilling Requests"
whatsnext:
 {
 "Performing System Maintenance": "/chainlink-nodes/resources/performing-
system-maintenance",
 "v2 Jobs": "/chainlink-nodes/oracle-jobs/jobs",
 "Security and Operation Best Practices": "/chainlink-nodes/resources/best-
security-practices",
 }
metadata:
 title: "Chainlink Node Operators: Fulfilling Requests"
 description: "Deploy your own operator contract and add jobs to your node so
that it can provide data to smart contracts."

```

```
import { Aside, CodeSample } from "@components"
```

```

<Aside type="note" title="Run a Chainlink node">
 This guide assumes you have a running Chainlink node. To learn how to run a
 node, see the Running a Chainlink Node
 locally guide.
</Aside>

```

You can use your Chainlink nodes to fulfill requests. This guide shows you how to deploy your own operator contract and add jobs to your node so that it can provide data to smart contracts.

Chainlink nodes can fulfill requests from open or unauthenticated APIs without the need for External Adapters as long as you've added the jobs to the node. For these requests, requesters supply the URL to the open API that they want each node to retrieve. The Chainlink node will use tasks to fulfill the request.

Some APIs require authentication by providing request headers for the operator's API key, which the Chainlink node supports. If you would like to provide access to an API that requires authentication, you must create a job that is specific for that API either using an external adapter or by using the parameters of the HTTP task.

## Requirements

Before you begin this guide, complete the following tasks to make sure you have all of the tools that you need:

- Set up MetaMask and obtain testnet LINK.
- Run a Chainlink Node.
- Fund the Ethereum address that your Chainlink node uses. You can find the address in the node Operator GUI under the Key Management configuration. The address of the node is the Regular type. You can obtain test ETH from several faucets. For this tutorial to work, you will have to fund the node's Ethereum address with Sepolia ETH. Here is an example:

```
!chainlink node fund address
```

## Address types

Your node works with several different types of addresses. Each address type has a specific function:

- Node address: This is the address for your Chainlink node wallet. The node requires native gas tokens at all times to respond to requests. For this example, the node uses Sepolia ETH. When you start a Chainlink node, it automatically generates this address. You can find this address on the Node Operator GUI under Key Management > EVM Chain Accounts.
- Oracle contract address: This is the address for contracts like Operator.sol that are deployed to a blockchain. Do not fund these addresses with native gas tokens such as ETH. When you make API call requests, the funds pass through this contract to interact with your Chainlink node. This will be the address that smart contract developers point to when they choose a node for an API call.
- Admin wallet address: This is the address that owns your Operator.sol contract addresses. If you're on OCR, this is the wallet address that receives LINK tokens.

## Set up your Operator contract

### Deploy your own Operator contract

1. Go to Remix and open the Operator.sol smart contract.

1. On the Compile tab, click the Compile button for Operator.sol. Remix automatically selects the compiler version and language from the pragma line unless you select a specific version manually.

1. On the Deploy and Run tab, configure the following settings:

- Select "Injected Provider" as your Environment. The Javascript VM environment cannot access your oracle node. Make sure your Metamask is connected to Sepolia testnet.

- Select the "Operator" contract from the Contract menu.
- Copy the LINK token contract address for the network you are using and paste it into the LINK field next to the Deploy button. For Sepolia, you can use this address:

```
text Sepolia
0x779877A7B0D9E8603169DdbD7836e478b4624789
```

- Copy the Admin wallet address into the OWNER field.

!The Deploy & Run transaction window showing Injected Web 3 selected and the address for your MetaMask wallet.

1. Click transact. MetaMask prompts you to confirm the transaction.

<Aside type="note" title="MetaMask doesn't pop up?">  
 If MetaMask does not prompt you and instead displays the error below, disable "Privacy Mode" in MetaMask. You can do this by clicking on your unique account icon at the top-right, then go to the Settings. Privacy Mode will be a switch near the bottom.  
 <br />  
 Error: Send transaction failed: Invalid address. If you use an injected provider, please check it is properly unlocked.  
 </Aside>

1. If the transaction is successful, a new address displays in the Deployed Contracts section.

!Screenshot showing the newly deployed contract.

1. Keep note of the Operator contract address. You need it later for your consuming contract.

Whitelist your node address in the Operator contract

1. In the Chainlink node GUI, find and copy the address of your chainlink node. see Requirements.

1. In Remix, call the setAuthorizedSenders function with the address of your node. Note the function expects an array.

!A screenshot showing all of the fields for the deployed contract in Remix.

1. Click the transact function to run it. Approve the transaction in MetaMask and wait for it to confirm on the blockchain.

1. Call isAuthorizedSender function with the address of your node to verify that your chainlink node address can call the operator contract. The function must return true.

!A screenshot showing Chainlink node whitelisted in Remix.

Add a job to the node

You will create a job that calls an OpenAPI , parses the response and then return a uint256.

1. In the Chainlink Operator UI on the Jobs tab, click New Job.

!The new job button.

1. Paste the job specification from above into the text field.

```
{/ prettier-ignore /}
<CodeSample src="samples/ChainlinkNodes/jobs/get-uint256.toml"/>
```

1. Replace YOUROPERATORCONTRACTADDRESS with the address of your deployed operator contract address from the previous steps.

1. Click Create Job. If the node creates the job successfully, a notice with the job number appears.

!A screenshot showing that the job is created successfully.

1. Click the job number to view the job details. You can also find the job listed on the Jobs tab in the Node Operators UI. Save the externalJobID value because you will need it later to tell your consumer contract what job ID to request from your node.

!A screenshot showing the External Job ID.

Create a request to your node

After you add jobs to your node, you can use the node to fulfill requests. This section shows what a requester does when they send requests to your node. It is also a way to test and make sure that your node is functioning correctly.

1. Open ATestnetConsumer.sol in Remix.

1. Note that setChainlinkToken(0x779877A7B0D9E8603169DdbD7836e478b4624789) is configured for Sepolia.

1. On the Compiler tab, click the Compile button for ATestnetConsumer.sol.

1. On the Deploy and Run tab, configure the following settings:

- Select Injected Provider as your environment. Make sure your metamask is connected to Sepolia.
- Select ATestnetConsumer from the Contract menu.

1. Click Deploy. MetaMask prompts you to confirm the transaction.

1. Fund the contract by sending LINK to the contract's address. See the Fund your contract page for instructions. The address for the ATestnetConsumer contract is on the list of your deployed contracts in Remix. You can fund your contract with 1 LINK.

1. After you fund the contract, create a request. Input your operator contract address and the job ID for the Get > Uint256 job into the requestEthereumPrice request method without dashes. The job ID is the externalJobID parameter, which you can find on your job's definition page in the Node Operators UI.

!Screenshot of the requestEthereumPrice function with the oracle address and job ID specified.

1. Click the transact button for the requestEthereumPrice function and approve the transaction in Metamask. The requestEthereumPrice function asks the node to retrieve uint256 data specifically from <https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=USD>.

1. After the transaction processes, you can see the details for the complete the job run the Runs page in the Node Operators UI.

!A screenshot of the task link

1. In Remix, click the `currentPrice` variable to see the current price updated on your consumer contract.

!A screenshot of the `currentPrice` button

## Withdrawing LINK

You can withdraw LINK from the operator contract. In Remix under the list of deployed contracts, click on your Operator contract and find the `withdraw` function in the function list. Note that only the admin (see Admin wallet address) can withdraw LINK.

!Remix Click Withdraw Button

Paste the address you want to withdraw to, and specify the amount of LINK that you want to withdraw. Then, click `withdraw`. Confirm the transaction in MetaMask when the popup appears.

# `index.mdx`:

```

section: nodeOperator
date: Last Modified
title: "Chainlink nodes v1"
isIndex: true
metadata:
 title: "Chainlink nodes v1"
 description: "How to run a Chainlink node which version is v1.x.x."
whatsnext:
 {
 "Running a Chainlink node locally": "/chainlink-nodes/v1/running-a-chainlink-node",
 "Fulfilling Requests": "/chainlink-nodes/v1/fulfilling-requests",
 "Configuring v1.x.x Chainlink nodes": "/chainlink-nodes/v1/configuration",
 }

```

This section provides specific information for running and configuring v1.x.x Chainlink nodes. First you will run a chainlink node that implements the basic request architecture model. Then you will fulfill requests initiated by a smart contract.

The `Configuring Nodes` page lists all the environment variables for configuring a v1.x.x Chainlink node.

# `node-config.mdx`:

```

section: nodeOperator
date: Last Modified
title: "Node Config (TOML)"

```

[//]: "Documentation generated from docs/.toml - DO NOT EDIT."

This document describes the TOML format for configuration.

See also: `Secrets Config`

Example

`toml`

Log.Level = 'debug'

[[EVM]]

ChainID = '1' Required

[[EVM.Nodes]]

Name = 'fake' Required

WSURL = 'wss://foo.bar/ws'

HTTPURL = 'https://foo.bar' Required

Global

toml

InsecureFastScript = false Default

RootDir = '/.chainlink' Default

ShutdownGracePeriod = '5s' Default

InsecureFastScript

⚠️, ADVANCED: Do not change this setting unless you know what you are doing.

toml

InsecureFastScript = false Default

InsecureFastScript causes all key stores to encrypt using "fast" script params instead. This is insecure and only useful for local testing. DO NOT ENABLE THIS IN PRODUCTION.

RootDir

toml

RootDir = '/.chainlink' Default

RootDir is the Chainlink node's root directory. This is the default directory for logging, database backups, cookies, and other misc Chainlink node files. Chainlink nodes will always ensure this directory has 700 permissions because it might contain sensitive data.

ShutdownGracePeriod

toml

ShutdownGracePeriod = '5s' Default

ShutdownGracePeriod is the maximum time allowed to shut down gracefully. If exceeded, the node will terminate immediately to avoid being SIGKILLED.

Feature

toml

[Feature]

FeedsManager = true Default

LogPoller = false Default

UICSAKeys = false Default

FeedsManager

toml

FeedsManager = true Default

FeedsManager enables the feeds manager service.

LogPoller

```
toml
LogPoller = false Default
```

LogPoller enables the log poller, an experimental approach to processing logs, required if also using Evm.UseForwarders or OCR2.

UICSAKeys

```
toml
UICSAKeys = false Default
```

UICSAKeys enables CSA Keys in the UI.

Database

```
toml
[Database]
DefaultIdleInTxSessionTimeout = '1h' Default
DefaultLockTimeout = '15s' Default
DefaultQueryTimeout = '10s' Default
LogQueries = false Default
MaxIdleConns = 10 Default
MaxOpenConns = 100 Default
MigrateOnStartup = true Default
```

DefaultIdleInTxSessionTimeout

```
toml
DefaultIdleInTxSessionTimeout = '1h' Default
```

DefaultIdleInTxSessionTimeout is the maximum time allowed for a transaction to be open and idle before timing out. See Postgres `idleintransactiontimeout` for more details.

DefaultLockTimeout

```
toml
DefaultLockTimeout = '15s' Default
```

DefaultLockTimeout is the maximum time allowed to wait for database lock of any kind before timing out. See Postgres `locktimeout` for more details.

DefaultQueryTimeout

```
toml
DefaultQueryTimeout = '10s' Default
```

DefaultQueryTimeout is the maximum time allowed for standard queries before timing out.

LogQueries



```
toml
LogQueries = false Default
```

LogQueries tells the Chainlink node to log database queries made using the default logger. SQL statements will be logged at debug level. Not all statements can be logged. The best way to get a true log of all SQL statements is to enable SQL statement logging on Postgres.

#### MaxIdleConns

```
toml
MaxIdleConns = 10 Default
```

MaxIdleConns configures the maximum number of idle database connections that the Chainlink node will keep open. Think of this as the baseline number of database connections per Chainlink node instance. Increasing this number can help to improve performance under database-heavy workloads.

Postgres has connection limits, so you must use caution when increasing this value. If you are running several instances of a Chainlink node or another application on a single database server, you might run out of Postgres connection slots if you raise this value too high.

#### MaxOpenConns

```
toml
MaxOpenConns = 100 Default
```

MaxOpenConns configures the maximum number of database connections that a Chainlink node will have open at any one time. Think of this as the maximum burst upper bound limit of database connections per Chainlink node instance. Increasing this number can help to improve performance under database-heavy workloads.

Postgres has connection limits, so you must use caution when increasing this value. If you are running several instances of a Chainlink node or another application on a single database server, you might run out of Postgres connection slots if you raise this value too high.

#### MigrateOnStartup

```
toml
MigrateOnStartup = true Default
```

MigrateOnStartup controls whether a Chainlink node will attempt to automatically migrate the database on boot. If you want more control over your database migration process, set this variable to false and manually migrate the database using the CLI migrate command instead.

#### Database.Backup

```
toml
[Database.Backup]
Mode = 'none' Default
Dir = 'test/backup/dir' Example
OnVersionUpgrade = true Default
Frequency = '1h' Default
```

As a best practice, take regular database backups in case of accidental data

loss. This best practice is especially important when you upgrade your Chainlink node to a new version. Chainlink nodes support automated database backups to make this process easier.

NOTE: Dumps can cause high load and massive database latencies, which will negatively impact the normal functioning of the Chainlink node. For this reason, it is recommended to set a URL and point it to a read replica if you enable automatic backups.

## Mode

```
toml
Mode = 'none' Default
```

Mode sets the type of automatic database backup, which can be one of none, lite, or full. If enabled, the Chainlink node will always dump a backup on every boot before running migrations. Additionally, it will automatically take database backups that overwrite the backup file for the given version at regular intervals if Frequency is set to a non-zero interval.

none - Disables backups.

lite - Dumps small tables including configuration and keys that are essential for the node to function, which excludes historical data like job runs, transaction history, etc.

full - Dumps the entire database.

It will write to a file like 'Dir'/backup/clbackup<VERSION>.dump. There is one backup dump file per version of the Chainlink node. If you upgrade the node, it will keep the backup taken right before the upgrade migration so you can restore to an older version if necessary.

## Dir

```
toml
Dir = 'test/backup/dir' Example
```

Dir sets the directory to use for saving the backup file. Use this if you want to save the backup file in a directory other than the default ROOT directory.

## OnVersionUpgrade

```
toml
OnVersionUpgrade = true Default
```

OnVersionUpgrade enables automatic backups of the database before running migrations, when you are upgrading to a new version.

## Frequency

```
toml
Frequency = '1h' Default
```

Frequency sets the interval for database dumps, if set to a positive duration and Mode is not none.

Set to 0 to disable periodic backups.

## Database.Listener

⚠ ADVANCED: Do not change these settings unless you know what you are

doing.

```
toml
[Database.Listener]
MaxReconnectDuration = '10m' Default
MinReconnectInterval = '1m' Default
FallbackPollInterval = '30s' Default
```

These settings control the postgres event listener.

MaxReconnectDuration

```
toml
MaxReconnectDuration = '10m' Default
```

MaxReconnectDuration is the maximum duration to wait between reconnect attempts.

MinReconnectInterval

```
toml
MinReconnectInterval = '1m' Default
```

MinReconnectInterval controls the duration to wait before trying to re-establish the database connection after connection loss. After each consecutive failure this interval is doubled, until MaxReconnectInterval is reached. Successfully completing the connection establishment procedure resets the interval back to MinReconnectInterval.

FallbackPollInterval

```
toml
FallbackPollInterval = '30s' Default
```

FallbackPollInterval controls how often clients should manually poll as a fallback in case the postgres event was missed/dropped.

Database.Lock

⚠️ ADVANCED: Do not change these settings unless you know what you are doing.

```
toml
[Database.Lock]
Enabled = true Default
LeaseDuration = '10s' Default
LeaseRefreshInterval = '1s' Default
```

Ideally, you should use a container orchestration system like Kubernetes to ensure that only one Chainlink node instance can ever use a specific Postgres database. However, some node operators do not have the technical capacity to do this. Common use cases run multiple Chainlink node instances in failover mode as recommended by our official documentation. The first instance takes a lock on the database and subsequent instances will wait trying to take this lock in case the first instance fails.

- If your nodes or applications hold locks open for several hours or days, Postgres is unable to complete internal cleanup tasks. The Postgres maintainers explicitly discourage holding locks open for long periods of time.

Because of the complications with advisory locks, Chainlink nodes with v2.0 and later only support lease locking mode. The lease locking mode works using the following process:

- Node A creates one row in the database with the client ID and updates it once per second.
- Node B spinlocks and checks periodically to see if the client ID is too old. If the client ID is not updated after a period of time, node B assumes that node A failed and takes over. Node B becomes the owner of the row and updates the client ID once per second.
- If node A comes back, it attempts to take out a lease, realizes that the database has been leased to another process, and exits the entire application immediately.

Enabled

```
toml
Enabled = true Default
```

Enabled enables the database lock.

LeaseDuration

```
toml
LeaseDuration = '10s' Default
```

LeaseDuration is how long the lease lock will last before expiring.

LeaseRefreshInterval

```
toml
LeaseRefreshInterval = '1s' Default
```

LeaseRefreshInterval determines how often to refresh the lease lock. Also controls how often a standby node will check to see if it can grab the lease.

TelemetryIngress

```
toml
[TelemetryIngress]
UniConn = true Default
Logging = false Default
BufferSize = 100 Default
MaxBatchSize = 50 Default
SendInterval = '500ms' Default
SendTimeout = '10s' Default
UseBatchSend = true Default
```

UniConn

```
toml
UniConn = true Default
```

UniConn toggles which ws connection style is used.

Logging

```
toml
Logging = false Default
```

Logging toggles verbose logging of the raw telemetry messages being sent.

#### BufferSize

```
toml
BufferSize = 100 Default
```

BufferSize is the number of telemetry messages to buffer before dropping new ones.

#### MaxBatchSize

```
toml
MaxBatchSize = 50 Default
```

MaxBatchSize is the maximum number of messages to batch into one telemetry request.

#### SendInterval

```
toml
SendInterval = '500ms' Default
```

SendInterval determines how often batched telemetry is sent to the ingress server.

#### SendTimeout

```
toml
SendTimeout = '10s' Default
```

SendTimeout is the max duration to wait for the request to complete when sending batch telemetry.

#### UseBatchSend

```
toml
UseBatchSend = true Default
```

UseBatchSend toggles sending telemetry to the ingress server using the batch client.

#### TelemetryIngress.Endpoints

```
toml
[[TelemetryIngress.Endpoints]] Example
Network = 'EVM' Example
ChainID = '111551111' Example
ServerPubKey = 'test-pub-key-111551111-evm' Example
URL = 'localhost-111551111-evm:9000' Example
```

#### Network

```
toml
Network = 'EVM' Example
```

Network aka EVM, Solana, Starknet

ChainID

```
toml
ChainID = '111551111' Example
```

ChainID of the network

ServerPubKey

```
toml
ServerPubKey = 'test-pub-key-111551111-evm' Example
```

ServerPubKey is the public key of the telemetry server.

URL

```
toml
URL = 'localhost-111551111-evm:9000' Example
```

URL is where to send telemetry.

AuditLogger

```
toml
[AuditLogger]
Enabled = false Default
ForwardToUrl = 'http://localhost:9898' Example
JsonWrapperKey = 'event' Example
Headers = ['Authorization: token', 'X-SomeOther-Header: value with spaces | and a bar+'] Example
```

Enabled

```
toml
Enabled = false Default
```

Enabled determines if this logger should be configured at all

ForwardToUrl

```
toml
ForwardToUrl = 'http://localhost:9898' Example
```

ForwardToUrl is where you want to forward logs to

JsonWrapperKey

```
toml
JsonWrapperKey = 'event' Example
```

JsonWrapperKey if set wraps the map of data under another single key to make parsing easier

Headers

```
toml
Headers = ['Authorization: token', 'X-SomeOther-Header: value with spaces | and
a bar+'] Example
```

Headers is the set of headers you wish to pass along with each request

Log

```
toml
[Log]
Level = 'info' Default
JSONConsole = false Default
UnixTS = false Default
```

Level

```
toml
Level = 'info' Default
```

Level determines both what is printed on the screen and what is written to the log file.

The available levels are:

- "debug": Useful for forensic debugging of issues.
- "info": High-level informational messages. (default)
- "warn": A mild error occurred that might require non-urgent action. Check these warnings semi-regularly to see if any of them require attention. These warnings usually happen due to factors outside of the control of the node operator. Examples: Unexpected responses from a remote API or misleading networking errors.
- "error": An unexpected error occurred during the regular operation of a well-maintained node. Node operators might need to take action to remedy this error. Check these regularly to see if any of them require attention. Examples: Use of deprecated configuration options or incorrectly configured settings that cause a job to fail.
- "crit": A critical error occurred. The node might be unable to function. Node operators should take immediate action to fix these errors. Examples: The node could not boot because a network socket could not be opened or the database became inaccessible.
- "panic": An exceptional error occurred that could not be handled. If the node is unresponsive, node operators should try to restart their nodes and notify the Chainlink team of a potential bug.
- "fatal": The node encountered an unrecoverable problem and had to exit.

JSONConsole

```
toml
JSONConsole = false Default
```

JSONConsole enables JSON logging. Otherwise, the log is saved in a human-friendly console format.

UnixTS

```
toml
UnixTS = false Default
```

UnixTS enables legacy unix timestamps.

Previous versions of Chainlink nodes wrote JSON logs with a unix timestamp. As of v1.1.0 and up, the default has changed to use ISO8601 timestamps for better readability.

#### Log.File

```
toml
[Log.File]
Dir = '/my/log/directory' Example
MaxSize = '5120mb' Default
MaxAgeDays = 0 Default
MaxBackups = 1 Default
```

#### Dir

```
toml
Dir = '/my/log/directory' Example
```

Dir sets the log directory. By default, Chainlink nodes write log data to \$ROOT/log.jsonl.

#### MaxSize

```
toml
MaxSize = '5120mb' Default
```

MaxSize determines the log file's max size in megabytes before file rotation. Having this not set will disable logging to disk. If your disk doesn't have enough disk space, the logging will pause and the application will log errors until space is available again.

Values must have suffixes with a unit like: 5120mb (5,120 megabytes). If no unit suffix is provided, the value defaults to b (bytes). The list of valid unit suffixes are:

- b (bytes)
- kb (kilobytes)
- mb (megabytes)
- gb (gigabytes)
- tb (terabytes)

#### MaxAgeDays

```
toml
MaxAgeDays = 0 Default
```

MaxAgeDays determines the log file's max age in days before file rotation. Keeping this config with the default value will not remove log files based on age.

#### MaxBackups

```
toml
MaxBackups = 1 Default
```

MaxBackups determines the maximum number of old log files to retain. Keeping this config with the default value retains all old log files. The MaxAgeDays



variable can still cause them to get deleted.

## WebServer

```
toml
[WebServer]
AuthenticationMethod = 'local' Default
AllowOrigins = 'http://localhost:3000,http://localhost:6688' Default
BridgeCacheTTL = '0s' Default
BridgeResponseURL = 'https://my-chainlink-node.example.com:6688' Example
HTTPWriteTimeout = '10s' Default
HTTPPort = 6688 Default
SecureCookies = true Default
SessionTimeout = '15m' Default
SessionReaperExpiration = '240h' Default
HTTPMaxSize = '32768b' Default
StartTimeout = '15s' Default
ListenIP = '0.0.0.0' Default
```

### AuthenticationMethod

```
toml
AuthenticationMethod = 'local' Default
```

AuthenticationMethod defines which pluggable auth interface to use for user login and role assumption. Options include 'local' and 'ldap'. See docs for more details

### AllowOrigins

```
toml
AllowOrigins = 'http://localhost:3000,http://localhost:6688' Default
```

AllowOrigins controls the URLs Chainlink nodes emit in the Allow-Origins header of its API responses. The setting can be a comma-separated list with no spaces. You might experience CORS issues if this is not set correctly.

You should set this to the external URL that you use to access the Chainlink UI.

You can set AllowOrigins = '' to allow the UI to work from any URL, but it is recommended for security reasons to make it explicit instead.

### BridgeCacheTTL

```
toml
BridgeCacheTTL = '0s' Default
```

BridgeCacheTTL controls the cache TTL for all bridge tasks to use old values in newer observations in case of intermittent failure. It's disabled by default.

### BridgeResponseURL

```
toml
BridgeResponseURL = 'https://my-chainlink-node.example.com:6688' Example
```

BridgeResponseURL defines the URL for bridges to send a response to. This must be set when using async external adapters.

Usually this will be the same as the URL/IP and port you use to connect to the

Chainlink UI.

HTTPWriteTimeout

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

toml

HTTPWriteTimeout = '10s' Default

HTTPWriteTimeout controls how long the Chainlink node's API server can hold a socket open for writing a response to an HTTP request. Sometimes, this must be increased for pprof.

HTTPPort

toml

HTTPPort = 6688 Default

HTTPPort is the port used for the Chainlink Node API, CLI, and GUI.

SecureCookies

toml

SecureCookies = true Default

SecureCookies requires the use of secure cookies for authentication. Set to false to enable standard HTTP requests along with TLSPort = 0.

SessionTimeout

toml

SessionTimeout = '15m' Default

SessionTimeout determines the amount of idle time to elapse before session cookies expire. This signs out GUI users from their sessions.

SessionReaperExpiration

toml

SessionReaperExpiration = '240h' Default

SessionReaperExpiration represents how long an API session lasts before expiring and requiring a new login.

HTTPMaxSize

toml

HTTPMaxSize = '32768b' Default

HTTPMaxSize defines the maximum size for HTTP requests and responses made by the node server.

StartTimeout

toml

StartTimeout = '15s' Default

StartTimeout defines the maximum amount of time the node will wait for a server to start.

ListenIP

```
toml
ListenIP = '0.0.0.0' Default
```

ListenIP specifies the IP to bind the HTTP server to

WebServer.LDAP

```
toml
[WebServer.LDAP]
ServerTLS = true Default
SessionTimeout = '15m0s' Default
QueryTimeout = '2m0s' Default
BaseUserAttr = 'uid' Default
BaseDN = 'dc=custom,dc=example,dc=com' Example
UsersDN = 'ou=users' Default
GroupsDN = 'ou=groups' Default
ActiveAttribute = '' Default
ActiveAttributeAllowedValue = '' Default
AdminUserGroupCN = 'NodeAdmins' Default
EditUserGroupCN = 'NodeEditors' Default
RunUserGroupCN = 'NodeRunners' Default
ReadUserGroupCN = 'NodeReadOnly' Default
UserApiTokenEnabled = false Default
UserAPITokenDuration = '240h0m0s' Default
UpstreamSyncInterval = '0s' Default
UpstreamSyncRateLimit = '2m0s' Default
```

Optional LDAP config if WebServer.AuthenticationMethod is set to 'ldap'  
LDAP queries are all parameterized to support custom LDAP 'dn', 'cn', and attributes

ServerTLS

```
toml
ServerTLS = true Default
```

ServerTLS defines the option to require the secure ldaps

SessionTimeout

```
toml
SessionTimeout = '15m0s' Default
```

SessionTimeout determines the amount of idle time to elapse before session cookies expire. This signs out GUI users from their sessions.

QueryTimeout

```
toml
QueryTimeout = '2m0s' Default
```

QueryTimeout defines how long queries should wait before timing out, defined in seconds

## BaseUserAttr

```
toml
BaseUserAttr = 'uid' Default
```

BaseUserAttr defines the base attribute used to populate LDAP queries such as "uid=\$", default is example

## BaseDN

```
toml
BaseDN = 'dc=custom,dc=example,dc=com' Example
```

BaseDN defines the base LDAP 'dn' search filter to apply to every LDAP query, replace example,com with the appropriate LDAP server's structure

## UsersDN

```
toml
UsersDN = 'ou=users' Default
```

UsersDN defines the 'dn' query to use when querying for the 'users' 'ou' group

## GroupsDN

```
toml
GroupsDN = 'ou=groups' Default
```

GroupsDN defines the 'dn' query to use when querying for the 'groups' 'ou' group

## ActiveAttribute

```
toml
ActiveAttribute = '' Default
```

ActiveAttribute is an optional user field to check truthiness for if a user is valid/active. This is only required if the LDAP provider lists inactive users as members of groups

## ActiveAttributeAllowedValue

```
toml
ActiveAttributeAllowedValue = '' Default
```

ActiveAttributeAllowedValue is the value to check against for the above optional user attribute

## AdminUserGroupCN

```
toml
AdminUserGroupCN = 'NodeAdmins' Default
```

AdminUserGroupCN is the LDAP 'cn' of the LDAP group that maps the core node's 'Admin' role

## EditUserGroupCN

toml  
EditUserGroupCN = 'NodeEditors' Default

EditUserGroupCN is the LDAP 'cn' of the LDAP group that maps the core node's 'Edit' role

RunUserGroupCN

toml  
RunUserGroupCN = 'NodeRunners' Default

RunUserGroupCN is the LDAP 'cn' of the LDAP group that maps the core node's 'Run' role

ReadUserGroupCN

toml  
ReadUserGroupCN = 'NodeReadOnly' Default

ReadUserGroupCN is the LDAP 'cn' of the LDAP group that maps the core node's 'Read' role

UserApiTokenEnabled

toml  
UserApiTokenEnabled = false Default

UserApiTokenEnabled enables the users to issue API tokens with the same access of their role

UserAPITokenDuration

toml  
UserAPITokenDuration = '240h0m0s' Default

UserAPITokenDuration is the duration of time an API token is active for before expiring

UpstreamSyncInterval

toml  
UpstreamSyncInterval = '0s' Default

UpstreamSyncInterval is the interval at which the background LDAP sync task will be called. A '0s' value disables the background sync being run on an interval. This check is already performed during login/logout actions, all sessions and API tokens stored in the local ldap tables are updated to match the remote server

UpstreamSyncRateLimit

toml  
UpstreamSyncRateLimit = '2m0s' Default

UpstreamSyncRateLimit defines a duration to limit the number of query/API calls to the upstream LDAP provider. It prevents the sync functionality from being called multiple times within the defined duration

## WebServer.RateLimit

```
toml
[WebServer.RateLimit]
Authenticated = 1000 Default
AuthenticatedPeriod = '1m' Default
Unauthenticated = 5 Default
UnauthenticatedPeriod = '20s' Default
```

### Authenticated

```
toml
Authenticated = 1000 Default
```

Authenticated defines the threshold to which authenticated requests get limited. More than this many authenticated requests per AuthenticatedRateLimitPeriod will be rejected.

### AuthenticatedPeriod

```
toml
AuthenticatedPeriod = '1m' Default
```

AuthenticatedPeriod defines the period to which authenticated requests get limited.

### Unauthenticated

```
toml
Unauthenticated = 5 Default
```

Unauthenticated defines the threshold to which unauthenticated requests get limited. More than this many unauthenticated requests per UnAuthenticatedRateLimitPeriod will be rejected.

### UnauthenticatedPeriod

```
toml
UnauthenticatedPeriod = '20s' Default
```

UnauthenticatedPeriod defines the period to which unauthenticated requests get limited.

## WebServer.MFA

```
toml
[WebServer.MFA]
RPID = 'localhost' Example
RPOrigin = 'http://localhost:6688/' Example
```

The Operator UI frontend supports enabling Multi Factor Authentication via Webauthn per account. When enabled, logging in will require the account password and a hardware or OS security key such as Yubikey. To enroll, log in to the operator UI and click the circle purple profile button at the top right and then click Register MFA Token. Tap your hardware security key or use the OS public key management feature to enroll a key. Next time you log in, this key will be required to authenticate.

## RPID

```
toml
RPID = 'localhost' Example
```

RPID is the FQDN of where the Operator UI is served. When serving locally, the value should be localhost.

## RPOrigin

```
toml
RPOrigin = 'http://localhost:6688/' Example
```

RPOrigin is the origin URL where WebAuthn requests initiate, including scheme and port. When serving locally, the value should be http://localhost:6688/.

## WebServer.TLS

```
toml
[WebServer.TLS]
CertPath = '/.cl/certs' Example
Host = 'tls-host' Example
KeyPath = '/home/$USER/.chainlink/tls/server.key' Example
HTTPSPort = 6689 Default
ForceRedirect = false Default
ListenIP = '0.0.0.0' Default
```

The TLS settings apply only if you want to enable TLS security on your Chainlink node.

## CertPath

```
toml
CertPath = '/.cl/certs' Example
```

CertPath is the location of the TLS certificate file.

## Host

```
toml
Host = 'tls-host' Example
```

Host is the hostname configured for TLS to be used by the Chainlink node. This is useful if you configured a domain name specific for your Chainlink node.

## KeyPath

```
toml
KeyPath = '/home/$USER/.chainlink/tls/server.key' Example
```

KeyPath is the location of the TLS private key file.

## HTTPSPort

```
toml
HTTPSPort = 6689 Default
```

HTTPSPort is the port used for HTTPS connections. Set this to 0 to disable HTTPS. Disabling HTTPS also relieves Chainlink nodes of the requirement for a TLS certificate.

#### ForceRedirect

```
toml
ForceRedirect = false Default
```

ForceRedirect forces TLS redirect for unencrypted connections.

#### ListenIP

```
toml
ListenIP = '0.0.0.0' Default
```

ListenIP specifies the IP to bind the HTTPS server to

#### JobPipeline

```
toml
[JobPipeline]
ExternalInitiatorsEnabled = false Default
MaxRunDuration = '10m' Default
MaxSuccessfulRuns = 10000 Default
ReaperInterval = '1h' Default
ReaperThreshold = '24h' Default
ResultWriteQueueDepth = 100 Default
VerboseLogging = true Default
```

#### ExternalInitiatorsEnabled

```
toml
ExternalInitiatorsEnabled = false Default
```

ExternalInitiatorsEnabled enables the External Initiator feature. If disabled, webhook jobs can ONLY be initiated by a logged-in user. If enabled, webhook jobs can be initiated by a whitelisted external initiator.

#### MaxRunDuration

```
toml
MaxRunDuration = '10m' Default
```

MaxRunDuration is the maximum time allowed for a single job run. If it takes longer, it will exit early and be marked errored. If set to zero, disables the time limit completely.

#### MaxSuccessfulRuns

```
toml
MaxSuccessfulRuns = 10000 Default
```

MaxSuccessfulRuns caps the number of completed successful runs per pipeline spec in the database. You can set it to zero as a performance optimisation; this will avoid saving any successful run.



Note this is not a hard cap, it can drift slightly larger than this but not by more than 5% or so.

#### ReaperInterval

```
toml
ReaperInterval = '1h' Default
```

ReaperInterval controls how often the job pipeline reaper will run to delete completed jobs older than ReaperThreshold, in order to keep database size manageable.

Set to 0 to disable the periodic reaper.

#### ReaperThreshold

```
toml
ReaperThreshold = '24h' Default
```

ReaperThreshold determines the age limit for job runs. Completed job runs older than this will be automatically purged from the database.

#### ResultWriteQueueDepth

⚠ i, ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
ResultWriteQueueDepth = 100 Default
```

ResultWriteQueueDepth controls how many writes will be buffered before subsequent writes are dropped, for jobs that write results asynchronously for performance reasons, such as OCR.

#### VerboseLogging

```
toml
VerboseLogging = true Default
```

VerboseLogging enables detailed logging of pipeline execution steps. This can be useful for debugging failed runs without relying on the UI or database.

You may disable if this results in excessive log volume.

#### JobPipeline.HTTPRequest

```
toml
[JobPipeline.HTTPRequest]
DefaultTimeout = '15s' Default
MaxSize = '32768' Default
```

#### DefaultTimeout

```
toml
DefaultTimeout = '15s' Default
```

DefaultTimeout defines the default timeout for HTTP requests made by http and bridge adapters.

## MaxSize

```
toml
MaxSize = '32768' Default
```

MaxSize defines the maximum size for HTTP requests and responses made by http and bridge adapters.

## FluxMonitor

```
toml
[FluxMonitor]
DefaultTransactionQueueDepth = 1 Default
SimulateTransactions = false Default
```

## DefaultTransactionQueueDepth

⚠️, ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
DefaultTransactionQueueDepth = 1 Default
```

DefaultTransactionQueueDepth controls the queue size for DropOldestStrategy in Flux Monitor. Set to 0 to use SendEvery strategy instead.

## SimulateTransactions

```
toml
SimulateTransactions = false Default
```

SimulateTransactions enables transaction simulation for Flux Monitor.

## OCR2

```
toml
[OCR2]
Enabled = false Default
ContractConfirmations = 3 Default
BlockchainTimeout = '20s' Default
ContractPollInterval = '1m' Default
ContractSubscribeInterval = '2m' Default
ContractTransmitterTransmitTimeout = '10s' Default
DatabaseTimeout = '10s' Default
KeyBundleID = '7a5f66bbe6594259325bf2b4f5b1a9c900000000000000000000000000000000'
Example
CaptureEATelemetry = false Default
CaptureAutomationCustomTelemetry = true Default
DefaultTransactionQueueDepth = 1 Default
SimulateTransactions = false Default
TraceLogging = false Default
```

## Enabled

```
toml
Enabled = false Default
```

Enabled enables OCR2 jobs.

## ContractConfirmations

```
toml
ContractConfirmations = 3 Default
```

ContractConfirmations is the number of block confirmations to wait for before enacting an on-chain configuration change. This value doesn't need to be very high (in particular, it does not need to protect against malicious re-orgs). Since configuration changes create some overhead, and mini-reorgs are fairly common, recommended values are between two and ten.

Malicious re-orgs are not any more of concern here than they are in blockchain applications in general: Since nodes check the contract for the latest config every `ContractConfigTrackerPollInterval.Seconds()`, they will come to a common view of the current config within any interval longer than that, as long as the latest setConfig transaction in the longest chain is stable. They will thus be able to continue reporting after the poll interval, unless an adversary is able to repeatedly re-org the transaction out during every poll interval, which would amount to the capability to censor any transaction.

Note that 1 confirmation implies that the transaction/event has been mined in one block.

0 confirmations would imply that the event would be recognised before it has even been mined, which is not currently supported.

e.g.

```
Current block height: 42
Changed in block height: 43
Contract config confirmations: 1
STILL PENDING
```

```
Current block height: 43
Changed in block height: 43
Contract config confirmations: 1
CONFIRMED
```

## BlockchainTimeout

```
toml
BlockchainTimeout = '20s' Default
```

BlockchainTimeout is the timeout for blockchain queries (mediated through `ContractConfigTracker` and `ContractTransmitter`). (This is necessary because an oracle's operations are serialized, so blocking forever on a chain interaction would break the oracle.)

## ContractPollInterval

```
toml
ContractPollInterval = '1m' Default
```

ContractPollInterval is the polling interval at which `ContractConfigTracker` is queried for updated on-chain configurations. Recommended values are between fifteen seconds and two minutes.

## ContractSubscribeInterval

```
toml
ContractSubscribeInterval = '2m' Default
```

ContractSubscribeInterval is the interval at which we try to establish a subscription on ContractConfigTracker if one doesn't exist. Recommended values are between two and five minutes.

ContractTransmitterTransmitTimeout

toml

ContractTransmitterTransmitTimeout = '10s' Default

ContractTransmitterTransmitTimeout is the timeout for ContractTransmitter.Transmit calls.

DatabaseTimeout

toml

DatabaseTimeout = '10s' Default

DatabaseTimeout is the timeout for database interactions. (This is necessary because an oracle's operations are serialized, so blocking forever on an observation would break the oracle.)

KeyBundleID

toml

KeyBundleID = '7a5f66bbe6594259325bf2b4f5b1a9c900000000000000000000000000000000'  
Example

KeyBundleID is a sha256 hexadecimal hash identifier.

CaptureEATelemetry

toml

CaptureEATelemetry = false Default

CaptureEATelemetry toggles collecting extra information from External Adapters

CaptureAutomationCustomTelemetry

toml

CaptureAutomationCustomTelemetry = true Default

CaptureAutomationCustomTelemetry toggles collecting automation specific telemetry

DefaultTransactionQueueDepth

toml

DefaultTransactionQueueDepth = 1 Default

DefaultTransactionQueueDepth controls the queue size for DropOldestStrategy in OCR2. Set to 0 to use SendEvery strategy instead.

SimulateTransactions

toml

SimulateTransactions = false Default

SimulateTransactions enables transaction simulation for OCR2.

TraceLogging

```
toml
TraceLogging = false Default
```

TraceLogging enables trace level logging.

OCR

```
toml
[OCR]
Enabled = false Default
ObservationTimeout = '5s' Default
BlockchainTimeout = '20s' Default
ContractPollInterval = '1m' Default
ContractSubscribeInterval = '2m' Default
DefaultTransactionQueueDepth = 1 Default
KeyBundleID = 'acdd42797a8b921b2910497badc5000600000000000000000000000000000000'
Example
SimulateTransactions = false Default
TransmitterAddress = '0xa0788FC17B1dEe36f057c42B6F373A34B014687e' Example
CaptureEATelemetry = false Default
TraceLogging = false Default
```

This section applies only if you are running off-chain reporting jobs.

Enabled

```
toml
Enabled = false Default
```

Enabled enables OCR jobs.

ObservationTimeout

```
toml
ObservationTimeout = '5s' Default
```

ObservationTimeout is the timeout for making observations using the DataSource.Observe method.  
(This is necessary because an oracle's operations are serialized, so blocking forever on an observation would break the oracle.)

BlockchainTimeout

```
toml
BlockchainTimeout = '20s' Default
```

BlockchainTimeout is the timeout for blockchain queries (mediated through ContractConfigTracker and ContractTransmitter).  
(This is necessary because an oracle's operations are serialized, so blocking forever on a chain interaction would break the oracle.)

ContractPollInterval

```
toml
ContractPollInterval = '1m' Default
```

ContractPollInterval is the polling interval at which ContractConfigTracker is queried for updated on-chain configurations. Recommended values are between fifteen seconds and two minutes.

ContractSubscribeInterval

```
toml
ContractSubscribeInterval = '2m' Default
```

ContractSubscribeInterval is the interval at which we try to establish a subscription on ContractConfigTracker if one doesn't exist. Recommended values are between two and five minutes.

DefaultTransactionQueueDepth

⚠️, ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
DefaultTransactionQueueDepth = 1 Default
```

DefaultTransactionQueueDepth controls the queue size for DropOldestStrategy in OCR. Set to 0 to use SendEvery strategy instead.

KeyBundleID

```
toml
KeyBundleID = 'acdd42797a8b921b2910497badc5000600000000000000000000000000000000'
Example
```

KeyBundleID is the default key bundle ID to use for OCR jobs. If you have an OCR job that does not explicitly specify a key bundle ID, it will fall back to this value.

SimulateTransactions

```
toml
SimulateTransactions = false Default
```

SimulateTransactions enables transaction simulation for OCR.

TransmitterAddress

```
toml
TransmitterAddress = '0xa0788FC17B1dEe36f057c42B6F373A34B014687e' Example
```

TransmitterAddress is the default sending address to use for OCR. If you have an OCR job that does not explicitly specify a transmitter address, it will fall back to this value.

CaptureEATelemetry

```
toml
CaptureEATelemetry = false Default
```

CaptureEATelemetry toggles collecting extra information from External Adapters

TraceLogging

```
toml
TraceLogging = false Default
```

TraceLogging enables trace level logging.

P2P

```
toml
[P2P]
IncomingMessageBufferSize = 10 Default
OutgoingMessageBufferSize = 10 Default
PeerID = '12D3KooWMoejJznyDuEk5aX6GvbjaG12UzeornPCBNzMRqdwrfJw' Example
TraceLogging = false Default
```

P2P has a versioned networking stack. Currently only [P2P.V2] is supported. All nodes in the OCR network should share the same networking stack.

IncomingMessageBufferSize

```
toml
IncomingMessageBufferSize = 10 Default
```

IncomingMessageBufferSize is the per-remote number of incoming messages to buffer. Any additional messages received on top of those already in the queue will be dropped.

OutgoingMessageBufferSize

```
toml
OutgoingMessageBufferSize = 10 Default
```

OutgoingMessageBufferSize is the per-remote number of outgoing messages to buffer. Any additional messages send on top of those already in the queue will displace the oldest.  
NOTE: OutgoingMessageBufferSize should be comfortably smaller than remote's IncomingMessageBufferSize to give the remote enough space to process them all in case we regained connection and now send a bunch at once

PeerID

```
toml
PeerID = '12D3KooWMoejJznyDuEk5aX6GvbjaG12UzeornPCBNzMRqdwrfJw' Example
```

PeerID is the default peer ID to use for OCR jobs. If unspecified, uses the first available peer ID.

TraceLogging

```
toml
TraceLogging = false Default
```

TraceLogging enables trace level logging.

## P2P.V2

```
toml
[P2P.V2]
Enabled = true Default
AnnounceAddresses = ['1.2.3.4:9999', '[a52d:0:a88:1274::abcd]:1337'] Example
DefaultBootstrappers =
['12D3KooWMHMLRQkgPbFSYHwD3NBuwtS1AmxhvKVUrcfyaGDASR4U@1.2.3.4:9999',
'12D3KooWM55u5Swtpw9r8aFLQHEtw7HR4t44GdNs654ej5gRs2Dh@example.com:1234'] Example
DeltaDial = '15s' Default
DeltaReconcile = '1m' Default
ListenAddresses = ['1.2.3.4:9999', '[a52d:0:a88:1274::abcd]:1337'] Example
```

### Enabled

```
toml
Enabled = true Default
```

Enabled enables P2P V2.

Note: V1.Enabled is true by default, so it must be set false in order to run V2 only.

### AnnounceAddresses

```
toml
AnnounceAddresses = ['1.2.3.4:9999', '[a52d:0:a88:1274::abcd]:1337'] Example
```

AnnounceAddresses is the addresses the peer will advertise on the network in host:port form as accepted by the TCP version of Go's net.Dial.

The addresses should be reachable by other nodes on the network. When attempting to connect to another node, a node will attempt to dial all of the other node's AnnounceAddresses in round-robin fashion.

### DefaultBootstrappers

```
toml
DefaultBootstrappers =
['12D3KooWMHMLRQkgPbFSYHwD3NBuwtS1AmxhvKVUrcfyaGDASR4U@1.2.3.4:9999',
'12D3KooWM55u5Swtpw9r8aFLQHEtw7HR4t44GdNs654ej5gRs2Dh@example.com:1234'] Example
```

DefaultBootstrappers is the default bootstrapper peers for libocr's v2 networking stack.

Oracle nodes typically only know each other's PeerIDs, but not their hostnames, IP addresses, or ports.

DefaultBootstrappers are special nodes that help other nodes discover each other's AnnounceAddresses so they can communicate.

Nodes continuously attempt to connect to bootstrappers configured in here. When a node wants to connect to another node

(which it knows only by PeerID, but not by address), it discovers the other node's AnnounceAddresses from communications

received from its DefaultBootstrappers or other discovered nodes. To facilitate discovery,

nodes will regularly broadcast signed announcements containing their PeerID and AnnounceAddresses.

### DeltaDial

```
toml
```



DeltaDial = '15s' Default

DeltaDial controls how far apart Dial attempts are

DeltaReconcile

toml

DeltaReconcile = '1m' Default

DeltaReconcile controls how often a Reconcile message is sent to every peer.

ListenAddresses

toml

ListenAddresses = ['1.2.3.4:9999', '[a52d:0:a88:1274::abcd]:1337'] Example

ListenAddresses is the addresses the peer will listen to on the network in host:port form as accepted by net.Listen(), but the host and port must be fully specified and cannot be empty. You can specify 0.0.0.0 (IPv4) or :: (IPv6) to listen on all interfaces, but that is not recommended.

Capabilities.ExternalRegistry

toml

[Capabilities.ExternalRegistry]

Address = '0x0' Example

NetworkID = 'evm' Default

ChainID = '1' Default

Address

toml

Address = '0x0' Example

Address is the address for the capabilities registry contract.

NetworkID

toml

NetworkID = 'evm' Default

NetworkID identifies the target network where the remote registry is located.

ChainID

toml

ChainID = '1' Default

ChainID identifies the target chain id where the remote registry is located.

Capabilities.Peering

toml

[Capabilities.Peering]

IncomingMessageBufferSize = 10 Default

OutgoingMessageBufferSize = 10 Default

PeerID = '12D3KooWMoejJznyDuEk5aX6GvbjaG12UzeornPCBNzMRqdwrFJw' Example  
TraceLogging = false Default

IncomingMessageBufferSize

toml  
IncomingMessageBufferSize = 10 Default

IncomingMessageBufferSize is the per-remote number of incoming messages to buffer. Any additional messages received on top of those already in the queue will be dropped.

OutgoingMessageBufferSize

toml  
OutgoingMessageBufferSize = 10 Default

OutgoingMessageBufferSize is the per-remote number of outgoing messages to buffer. Any additional messages send on top of those already in the queue will displace the oldest.  
NOTE: OutgoingMessageBufferSize should be comfortably smaller than remote's IncomingMessageBufferSize to give the remote enough space to process them all in case we regained connection and now send a bunch at once

PeerID

toml  
PeerID = '12D3KooWMoejJznyDuEk5aX6GvbjaG12UzeornPCBNzMRqdwrFJw' Example

PeerID is the default peer ID to use for OCR jobs. If unspecified, uses the first available peer ID.

TraceLogging

toml  
TraceLogging = false Default

TraceLogging enables trace level logging.

Capabilities.Peering.V2

toml  
[Capabilities.Peering.V2]  
Enabled = false Default  
AnnounceAddresses = ['1.2.3.4:9999', '[a52d:0:a88:1274::abcd]:1337'] Example  
DefaultBootstrappers =  
['12D3KooWMHMLQkgPbFSYHwD3NBuwtS1AmxhvKVUrcfyaGDASR4U@1.2.3.4:9999',  
'12D3KooWM55u5Swtpw9r8aFLQHEtw7HR4t44GdNs654ej5gRs2Dh@example.com:1234'] Example  
DeltaDial = '15s' Default  
DeltaReconcile = '1m' Default  
ListenAddresses = ['1.2.3.4:9999', '[a52d:0:a88:1274::abcd]:1337'] Example

Enabled

toml  
Enabled = false Default

Enabled enables P2P V2.

#### AnnounceAddresses

toml

```
AnnounceAddresses = ['1.2.3.4:9999', '[a52d:0:a88:1274::abcd]:1337'] Example
```

AnnounceAddresses is the addresses the peer will advertise on the network in host:port form as accepted by the TCP version of Go's net.Dial. The addresses should be reachable by other nodes on the network. When attempting to connect to another node, a node will attempt to dial all of the other node's AnnounceAddresses in round-robin fashion.

#### DefaultBootstrappers

toml

```
DefaultBootstrappers =
['12D3KooWMHMLRQkgPbFSYHwD3NBuwtS1AmxhvKVUrcfyaGDASR4U@1.2.3.4:9999',
'12D3KooWM55u5Swtpw9r8aFLQHETw7HR4t44GdNs654ej5gRs2Dh@example.com:1234'] Example
```

DefaultBootstrappers is the default bootstrapper peers for libocr's v2 networking stack.

Oracle nodes typically only know each other's PeerIDs, but not their hostnames, IP addresses, or ports. DefaultBootstrappers are special nodes that help other nodes discover each other's AnnounceAddresses so they can communicate. Nodes continuously attempt to connect to bootstrappers configured in here. When a node wants to connect to another node (which it knows only by PeerID, but not by address), it discovers the other node's AnnounceAddresses from communications received from its DefaultBootstrappers or other discovered nodes. To facilitate discovery, nodes will regularly broadcast signed announcements containing their PeerID and AnnounceAddresses.

#### DeltaDial

toml

```
DeltaDial = '15s' Default
```

DeltaDial controls how far apart Dial attempts are

#### DeltaReconcile

toml

```
DeltaReconcile = '1m' Default
```

DeltaReconcile controls how often a Reconcile message is sent to every peer.

#### ListenAddresses

toml

```
ListenAddresses = ['1.2.3.4:9999', '[a52d:0:a88:1274::abcd]:1337'] Example
```

ListenAddresses is the addresses the peer will listen to on the network in host:port form as accepted by net.Listen(), but the host and port must be fully specified and cannot be empty. You can

specify 0.0.0.0 (IPv4) or :: (IPv6) to listen on all interfaces, but that is not recommended.

## Keeper

```
toml
[Keeper]
DefaultTransactionQueueDepth = 1 Default
GasPriceBufferPercent = 20 Default
GasTipCapBufferPercent = 20 Default
BaseFeeBufferPercent = 20 Default
MaxGracePeriod = 100 Default
TurnLookBack = 1000 Default
```

### DefaultTransactionQueueDepth

⚠️ **ADVANCED:** Do not change this setting unless you know what you are doing.

```
toml
DefaultTransactionQueueDepth = 1 Default
```

DefaultTransactionQueueDepth controls the queue size for DropOldestStrategy in Keeper. Set to 0 to use SendEvery strategy instead.

### GasPriceBufferPercent

⚠️ **ADVANCED:** Do not change this setting unless you know what you are doing.

```
toml
GasPriceBufferPercent = 20 Default
```

GasPriceBufferPercent specifies the percentage to add to the gas price used for checking whether to perform an upkeep. Only applies in legacy mode (EIP-1559 off).

### GasTipCapBufferPercent

⚠️ **ADVANCED:** Do not change this setting unless you know what you are doing.

```
toml
GasTipCapBufferPercent = 20 Default
```

GasTipCapBufferPercent specifies the percentage to add to the gas price used for checking whether to perform an upkeep. Only applies in EIP-1559 mode.

### BaseFeeBufferPercent

⚠️ **ADVANCED:** Do not change this setting unless you know what you are doing.

```
toml
BaseFeeBufferPercent = 20 Default
```

BaseFeeBufferPercent specifies the percentage to add to the base fee used for checking whether to perform an upkeep. Applies only in EIP-1559 mode.

### MaxGracePeriod

⚠️ **ADVANCED:** Do not change this setting unless you know what you are doing.

```
toml
MaxGracePeriod = 100 Default
```

MaxGracePeriod is the maximum number of blocks that a keeper will wait after performing an upkeep before it resumes checking that upkeep

TurnLookBack

```
toml
TurnLookBack = 1000 Default
```

TurnLookBack is the number of blocks in the past to look back when getting a block for a turn.

Keeper.Registry

```
toml
[Keeper.Registry]
CheckGasOverhead = 200000 Default
PerformGasOverhead = 300000 Default
SyncInterval = '30m' Default
MaxPerformDataSize = 5000 Default
SyncUpkeepQueueSize = 10 Default
```

CheckGasOverhead

⚠️ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
CheckGasOverhead = 200000 Default
```

CheckGasOverhead is the amount of extra gas to provide checkUpkeep() calls to account for the gas consumed by the keeper registry.

PerformGasOverhead

⚠️ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
PerformGasOverhead = 300000 Default
```

PerformGasOverhead is the amount of extra gas to provide performUpkeep() calls to account for the gas consumed by the keeper registry

SyncInterval

⚠️ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
SyncInterval = '30m' Default
```

SyncInterval is the interval in which the RegistrySynchronizer performs a full sync of the keeper registry contract it is tracking.

MaxPerformDataSize

⚠️ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
MaxPerformDataSize = 5000 Default
```

MaxPerformDataSize is the max size of perform data.

SyncUpkeepQueueSize

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
SyncUpkeepQueueSize = 10 Default
```

SyncUpkeepQueueSize represents the maximum number of upkeeps that can be synced in parallel.

AutoPprof

```
toml
[AutoPprof]
Enabled = false Default
ProfileRoot = 'prof/root' Example
PollInterval = '10s' Default
GatherDuration = '10s' Default
GatherTraceDuration = '5s' Default
MaxProfileSize = '100mb' Default
CPUProfileRate = 1 Default
MemProfileRate = 1 Default
BlockProfileRate = 1 Default
MutexProfileFraction = 1 Default
MemThreshold = '4gb' Default
GoroutineThreshold = 5000 Default
```

The Chainlink node is equipped with an internal "nurse" service that can perform automatic pprof profiling when the certain resource thresholds are exceeded, such as memory and goroutine count. These profiles are saved to disk to facilitate fine-grained debugging of performance-related issues. In general, if you notice that your node has begun to accumulate profiles, forward them to the Chainlink team.

To learn more about these profiles, read the [Profiling Go programs with pprof](#) guide.

Enabled

```
toml
Enabled = false Default
```

Enabled enables the automatic profiling service.

ProfileRoot

```
toml
ProfileRoot = 'prof/root' Example
```

ProfileRoot sets the location on disk where pprof profiles will be stored. Defaults to RootDir.

PollInterval

```
toml
PollInterval = '10s' Default
```

PollInterval is the interval at which the node's resources are checked.

GatherDuration

```
toml
GatherDuration = '10s' Default
```

GatherDuration is the duration for which profiles are gathered when profiling starts.

GatherTraceDuration

```
toml
GatherTraceDuration = '5s' Default
```

GatherTraceDuration is the duration for which traces are gathered when profiling is kicked off. This is separately configurable because traces are significantly larger than other types of profiles.

MaxProfileSize

```
toml
MaxProfileSize = '100mb' Default
```

MaxProfileSize is the maximum amount of disk space that profiles may consume before profiling is disabled.

CPUProfileRate

```
toml
CPUProfileRate = 1 Default
```

CPUProfileRate sets the rate for CPU profiling. See <https://pkg.go.dev/runtime#SetCPUProfileRate>.

MemProfileRate

```
toml
MemProfileRate = 1 Default
```

MemProfileRate sets the rate for memory profiling. See <https://pkg.go.dev/runtime#pkg-variables>.

BlockProfileRate

```
toml
BlockProfileRate = 1 Default
```

BlockProfileRate sets the fraction of blocking events for goroutine profiling. See <https://pkg.go.dev/runtime#SetBlockProfileRate>.

MutexProfileFraction

```
toml
```

MutexProfileFraction = 1 Default

MutexProfileFraction sets the fraction of contention events for mutex profiling. See <https://pkg.go.dev/runtime#SetMutexProfileFraction>.

MemThreshold

```
toml
MemThreshold = '4gb' Default
```

MemThreshold sets the maximum amount of memory the node can actively consume before profiling begins.

GoroutineThreshold

```
toml
GoroutineThreshold = 5000 Default
```

GoroutineThreshold is the maximum number of actively-running goroutines the node can spawn before profiling begins.

Pyroscope

```
toml
[Pyroscope]
ServerAddress = 'http://localhost:4040' Example
Environment = 'mainnet' Default
```

ServerAddress

```
toml
ServerAddress = 'http://localhost:4040' Example
```

ServerAddress sets the address that will receive the profile logs. It enables the profiling service.

Environment

```
toml
Environment = 'mainnet' Default
```

Environment sets the target environment tag in which profiles will be added to.

Sentry

```
toml
[Sentry]
Debug = false Default
DSN = 'sentry-dsn' Example
Environment = 'my-custom-env' Example
Release = 'v1.2.3' Example
```

Debug

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
```



Debug = false Default

Debug enables printing of Sentry SDK debug messages.

DSN

```
toml
DSN = 'sentry-dsn' Example
```

DSN is the data source name where events will be sent. Sentry is completely disabled if this is left blank.

Environment

```
toml
Environment = 'my-custom-env' Example
```

Environment overrides the Sentry environment to the given value. Otherwise autodetects between dev/prod.

Release

```
toml
Release = 'v1.2.3' Example
```

Release overrides the Sentry release to the given value. Otherwise uses the compiled-in version number.

Insecure

```
toml
[Insecure]
DevWebServer = false Default
OCRDevelopmentMode = false Default
InfiniteDepthQueries = false Default
DisableRateLimiting = false Default
```

Insecure config family is only allowed in development builds.

DevWebServer

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
DevWebServer = false Default
```

DevWebServer skips secure configuration for webserver AllowedHosts, SSL, etc.

OCRDevelopmentMode

```
toml
OCRDevelopmentMode = false Default
```

OCRDevelopmentMode run OCR in development mode.

InfiniteDepthQueries

```
toml
InfiniteDepthQueries = false Default
```

InfiniteDepthQueries skips graphql query depth limit checks.

DisableRateLimiting

```
toml
DisableRateLimiting = false Default
```

DisableRateLimiting skips ratelimiting on asset requests.

Tracing

```
toml
[Tracing]
Enabled = false Default
CollectorTarget = 'localhost:4317' Example
NodeID = 'NodeID' Example
SamplingRatio = 1.0 Example
Mode = 'tls' Default
TLSCertPath = '/path/to/cert.pem' Example
```

Enabled

```
toml
Enabled = false Default
```

Enabled turns trace collection on or off. On requires an OTEL Tracing Collector.

CollectorTarget

```
toml
CollectorTarget = 'localhost:4317' Example
```

CollectorTarget is the logical address of the OTEL Tracing Collector.

NodeID

```
toml
NodeID = 'NodeID' Example
```

NodeID is an unique name for this node relative to any other node traces are collected for.

SamplingRatio

```
toml
SamplingRatio = 1.0 Example
```

SamplingRatio is the ratio of traces to sample for this node.

Mode

```
toml
Mode = 'tls' Default
```

Mode is a string value. tls or unencrypted are the only values allowed. If set to unencrypted, TLSCertPath can be unset, meaning traces will be sent over plaintext to the collector.

TLSCertPath

```
toml
TLSCertPath = '/path/to/cert.pem' Example
```

TLSCertPath is the file path to the TLS certificate used for secure communication with an OTEL Tracing Collector.

Tracing.Attributes

```
toml
[Tracing.Attributes]
env = 'test' Example
```

Tracing.Attributes are user specified key-value pairs to associate in the context of the traces

env

```
toml
env = 'test' Example
```

env is an example user specified key-value pair

Mercury

```
toml
[Mercury]
VerboseLogging = false Default
```

VerboseLogging

```
toml
VerboseLogging = false Default
```

VerboseLogging enables detailed logging of mercury/LL0 operations. These logs can be expensive since they may serialize large structs, so they are disabled by default.

Mercury.Cache

```
toml
[Mercury.Cache]
LatestReportTTL = "1s" Default
MaxStaleAge = "1h" Default
LatestReportDeadline = "5s" Default
```

Mercury.Cache controls settings for the price retrieval cache querying a mercury server

LatestReportTTL

```
toml
```

LatestReportTTL = "1s" Default

LatestReportTTL controls how "stale" we will allow a price to be e.g. if set to 1s, a new price will always be fetched if the last result was from 1 second ago or older.

Another way of looking at it is such: the cache will never return a price that was queried from now-LatestReportTTL or before.

Setting to zero disables caching entirely.

MaxStaleAge

toml

MaxStaleAge = "1h" Default

MaxStaleAge is that maximum amount of time that a value can be stale before it is deleted from the cache (a form of garbage collection).

This should generally be set to something much larger than LatestReportTTL. Setting to zero disables garbage collection.

LatestReportDeadline

toml

LatestReportDeadline = "5s" Default

LatestReportDeadline controls how long to wait for a response from the mercury server before retrying. Setting this to zero will wait indefinitely.

Mercury.TLS

toml

[Mercury.TLS]

CertFile = "/path/to/client/certs.pem" Example

Mercury.TLS controls client settings for when the node talks to traditional web servers or load balancers.

CertFile

toml

CertFile = "/path/to/client/certs.pem" Example

CertFile is the path to a PEM file of trusted root certificate authority certificates

Mercury.Transmitter

toml

[Mercury.Transmitter]

TransmitQueueMaxSize = 10000 Default

TransmitTimeout = "5s" Default

Mercury.Transmitter controls settings for the mercury transmitter

TransmitQueueMaxSize

```
toml
TransmitQueueMaxSize = 10000 Default
```

TransmitQueueMaxSize controls the size of the transmit queue. This is scoped per OCR instance. If the queue is full, the transmitter will start dropping the oldest messages in order to make space.

This is useful if mercury server goes offline and the node needs to buffer transmissions.

```
TransmitTimeout
```

```
toml
TransmitTimeout = "5s" Default
```

TransmitTimeout controls how long the transmitter will wait for a response when sending a message to the mercury server, before aborting and considering the transmission to be failed.

EVM

EVM defaults depend on ChainID:

Ethereum Mainnet (1)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x514910771AF9Ca656af840dff83E8264EcF986CA'
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.1 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
OperatorFactoryAddress = '0x3E64Cd889482443324F91bFA9c84fE72A511f48A'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
```

```
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 4
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 50
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 10500000
```

Ethereum Ropsten (3)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x20fE562d797A42Dcb3399062AE9546cd06f63280'
```

```
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.1 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 4
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 50
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
```

```
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

#### Ethereum Rinkeby (4)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x01BE23585060835E02B77ef475b0Cc51aA1e0709'
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.1 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
```



```
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 4
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 50

[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Ethereum Goerli (5)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x326C977E6efc84E512bB9C30f76E30c160eD06FB'
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
```

```
MinContractPayment = '0.1 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 4
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 50
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Optimism Mainnet (10)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'optimismBedrock'
FinalityDepth = 200
FinalityTagEnabled = false
LinkContractAddress = '0x350a791Bfc2C21F9Ed5d10980Dad2e2638ffa7f6'
LogBackfillBatchSize = 1000
LogPollInterval = '2s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '40s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '30s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 wei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '100 wei'
BumpPercent = 20
```

```
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 300
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 6500000
```

RSK Mainnet (30)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x14AdaE34beF7ca957Ce2dDe5ADD97ea050123827'
LogBackfillBatchSize = 1000
LogPollInterval = '30s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
```

FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '50 mwei'  
PriceMax = '50 gwei'  
PriceMin = '0'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 mwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 8  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 4  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'

ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

RSK Testnet (31)

toml  
AutoCreateKey = true  
BlockBackfillDepth = 10  
BlockBackfillSkip = false  
FinalityDepth = 50  
FinalityTagEnabled = false  
LinkContractAddress = '0x8bBbd80981FE76d44854D8DF305e8985c19f0e78'  
LogBackfillBatchSize = 1000  
LogPollInterval = '30s'  
LogKeepBlocksDepth = 100000  
LogPrunePageSize = 0  
BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 3  
MinContractPayment = '0.001 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '3m0s'  
RPCDefaultBatchSize = 250  
RPCBlockQueryDelay = 1  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '50 mwei'  
PriceMax = '50 gwei'  
PriceMin = '0'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 mwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 8  
CheckInclusionBlocks = 12

CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 4  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Ethereum Kovan (42)

toml  
AutoCreateKey = true  
BlockBackfillDepth = 10  
BlockBackfillSkip = false  
FinalityDepth = 50  
FinalityTagEnabled = false  
LinkContractAddress = '0xa36085F69e2889c224210F603D836748e7dC0088'  
LogBackfillBatchSize = 1000  
LogPollInterval = '15s'  
LogKeepBlocksDepth = 100000  
LogPrunePageSize = 0  
BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 3  
MinContractPayment = '0.1 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '3m0s'  
OperatorFactoryAddress = '0x8007e24251b1D2Fc518Eb843A701d9cD21fe0aA3'  
RPCDefaultBatchSize = 250  
RPCBlockQueryDelay = 1  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '20 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '1 gwei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 4  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 50

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 4  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000



## BSC Mainnet (56)

toml

```
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x404460C6A5EdE2D891e8297795264fDe62ADBB75'
LogBackfillBatchSize = 1000
LogPollInterval = '3s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 2
FinalizedBlockOffset = 0
```

[Transactions]

```
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

[Transactions.AutoPurge]

```
Enabled = false
```

[BalanceMonitor]

```
Enabled = true
```

[GasEstimator]

```
Mode = 'BlockHistory'
PriceDefault = '5 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 5
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

[GasEstimator.BlockHistory]

```
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

[HeadTracker]

```
HistoryDepth = 100
```

```
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '2s'
DatabaseTimeout = '2s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '500ms'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

OKX Testnet (65)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

OKX Mainnet (66)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
```

```
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
```

```
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

BSC Testnet (97)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x84b9B910527Ad5C03A9Ca831909E21e236EA7b06'
LogBackfillBatchSize = 1000
LogPollInterval = '3s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 2
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '5 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
```

```
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 5
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = false
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '2s'
DatabaseTimeout = '2s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '500ms'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Gnosis Mainnet (100)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'gnosis'
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0xE2e73A1c69ecF83F464EFCE6A5be353a37cA09b2'
LogBackfillBatchSize = 1000
LogPollInterval = '5s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
```

```
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '1 gwei'
PriceMax = '500 gwei'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Heco Mainnet (128)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x404460C6A5EdE2D891e8297795264fDe62ADBB75'
LogBackfillBatchSize = 1000
LogPollInterval = '3s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 2
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '5 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 5
```



```
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '2s'
DatabaseTimeout = '2s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '500ms'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

## Polygon Mainnet (137)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 500
FinalityTagEnabled = false
LinkContractAddress = '0xb0897686c545045aFc77CF20eC7A532E3120E0F1'
LogBackfillBatchSize = 1000
LogPollInterval = '1s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 5
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 100
RPCBlockQueryDelay = 10
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 5000
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '30 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '30 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '20 gwei'
BumpPercent = 20
BumpThreshold = 5
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 2000
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
```

```
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

XLayer Sepolia (195)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'xlayer'
FinalityDepth = 500
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '30s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '12m0s'
RPCDefaultBatchSize = 100
RPCBlockQueryDelay = 15
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '3m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 mwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '20 mwei'
BumpPercent = 40
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
```

BatchSize = 25  
BlockHistorySize = 12  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 2000  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

## XLayer Mainnet (196)

toml  
AutoCreateKey = true  
BlockBackfillDepth = 10  
BlockBackfillSkip = false  
ChainType = 'xlayer'  
FinalityDepth = 500  
FinalityTagEnabled = false  
LogBackfillBatchSize = 1000  
LogPollInterval = '30s'  
LogKeepBlocksDepth = 100000  
LogPrunePageSize = 0  
BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 1  
MinContractPayment = '0.00001 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '6m0s'  
RPCDefaultBatchSize = 100  
RPCBlockQueryDelay = 15  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'

ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '3m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '20 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '100 mwei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '100 mwei'  
BumpPercent = 40  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 12  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 2000  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Fantom Mainnet (250)

toml

```
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x6F43FF82CCA38001B6699a8AC47A2d0E66939407'
LogBackfillBatchSize = 1000
LogPollInterval = '1s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 2
FinalizedBlockOffset = 0
```

[Transactions]

```
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

[Transactions.AutoPurge]

```
Enabled = false
```

[BalanceMonitor]

```
Enabled = true
```

[GasEstimator]

```
Mode = 'SuggestedPrice'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

[GasEstimator.BlockHistory]

```
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 3800000
```

Kroma Mainnet (255)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'kroma'
FinalityDepth = 400
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '2s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '40s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '30s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true

[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 wei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '100 wei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 400
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 5400000

zkSync Goerli (280)

toml
AutoCreateKey = true
```



BlockBackfillDepth = 10  
BlockBackfillSkip = false  
ChainType = 'zksync'  
FinalityDepth = 10  
FinalityTagEnabled = false  
LogBackfillBatchSize = 1000  
LogPollInterval = '5s'  
LogKeepBlocksDepth = 100000  
LogPrunePageSize = 0  
BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 1  
MinContractPayment = '0.00001 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '1m0s'  
RPCDefaultBatchSize = 250  
RPCBlockQueryDelay = 1  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '20 gwei'  
PriceMax = '18.446744073709551615 ether'  
PriceMin = '0'  
LimitDefault = 100000000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 8  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 50  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]

```
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

zkSync Sepolia (300)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'zksync'
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '5s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '1m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax = '18.446744073709551615 ether'
PriceMin = '0'
```

```
LimitDefault = 1000000000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 50
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

zkSync Mainnet (324)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'zksync'
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '5s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
```

```
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '1m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0

[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]
Enabled = false

[BalanceMonitor]
Enabled = true

[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax = '18.446744073709551615 ether'
PriceMin = '0'
LimitDefault = 1000000000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 50
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Optimism Goerli (420)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'optimismBedrock'
FinalityDepth = 200
FinalityTagEnabled = false
LinkContractAddress = '0xdc2CC710e42857672E7907CF474a69B63B93089f'
LogBackfillBatchSize = 1000
LogPollInterval = '2s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '40s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '30s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 wei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '100 wei'
BumpPercent = 20
BumpThreshold = 3
```

```
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 60
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 300
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 6500000
```

Metis Rinkeby (588)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'metis'
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'SuggestedPrice'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '0'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 0
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
```

```
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Klaytn Testnet (1001)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'SuggestedPrice'
PriceDefault = '750 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 5
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
```



BlockHistorySize = 8  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Metis Mainnet (1088)

toml  
AutoCreateKey = true  
BlockBackfillDepth = 10  
BlockBackfillSkip = false  
ChainType = 'metis'  
FinalityDepth = 10  
FinalityTagEnabled = false  
LogBackfillBatchSize = 1000  
LogPollInterval = '15s'  
LogKeepBlocksDepth = 100000  
LogPrunePageSize = 0  
BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 1  
MinContractPayment = '0.00001 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '0s'  
RPCDefaultBatchSize = 250  
RPCBlockQueryDelay = 1  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'

ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'SuggestedPrice'  
PriceDefault = '20 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '0'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 0  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 10  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 54000000

## Polygon Zkevm Mainnet (1101)

toml

```
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'zkevm'
FinalityDepth = 500
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '30s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '6m0s'
RPCDefaultBatchSize = 100
RPCBlockQueryDelay = 15
FinalizedBlockOffset = 0
```

[Transactions]

```
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '3m0s'
```

[Transactions.AutoPurge]

```
Enabled = false
```

[BalanceMonitor]

```
Enabled = true
```

[GasEstimator]

```
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '100 mwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '100 mwei'
BumpPercent = 40
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

[GasEstimator.BlockHistory]

```
BatchSize = 25
BlockHistorySize = 12
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

[HeadTracker]

```
HistoryDepth = 2000
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

WeMix Mainnet (1111)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'wemix'
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '3s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
```

Enabled = true

[GasEstimator]

Mode = 'BlockHistory'

PriceDefault = '20 gwei'

PriceMax =

'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'

PriceMin = '1 gwei'

LimitDefault = 500000

LimitMax = 500000

LimitMultiplier = '1'

LimitTransfer = 21000

BumpMin = '5 gwei'

BumpPercent = 20

BumpThreshold = 3

EIP1559DynamicFees = true

FeeCapDefault = '100 gwei'

TipCapDefault = '100 gwei'

TipCapMin = '1 wei'

[GasEstimator.BlockHistory]

BatchSize = 25

BlockHistorySize = 8

CheckInclusionBlocks = 12

CheckInclusionPercentile = 90

TransactionPercentile = 60

[HeadTracker]

HistoryDepth = 100

MaxBufferSize = 3

SamplingInterval = '1s'

MaxAllowedFinalityDepth = 10000

FinalityTagBypass = true

[NodePool]

PollFailureThreshold = 5

PollInterval = '10s'

SelectionMode = 'HighestHead'

SyncThreshold = 5

LeaseDuration = '0s'

NodeIsSyncingEnabled = false

FinalizedBlockPollInterval = '5s'

EnforceRepeatableRead = false

DeathDeclarationDelay = '10s'

[OCR]

ContractConfirmations = 1

ContractTransmitterTransmitTimeout = '10s'

DatabaseTimeout = '10s'

DeltaCOverride = '168h0m0s'

DeltaCJitterOverride = '1h0m0s'

ObservationGracePeriod = '1s'

[OCR2]

[OCR2.Automation]

GasLimit = 5400000

WeMix Testnet (1112)

toml

AutoCreateKey = true

BlockBackfillDepth = 10

```
BlockBackfillSkip = false
ChainType = 'wemix'
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '3s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '100 gwei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = false
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Simulated (1337)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '100'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '0s'
ResendAfterThreshold = '0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'FixedPrice'
PriceDefault = '20 gwei'
PriceMax = '100 micro'
PriceMin = '0'
```

```
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 0
EIP1559DynamicFees = false
FeeCapDefault = '100 micro'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 10
MaxBufferSize = 100
SamplingInterval = '0s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Polygon Zkevm Goerli (1442)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'zkevm'
FinalityDepth = 500
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '30s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
```



```
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '12m0s'
RPCDefaultBatchSize = 100
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '3m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '50 mwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '20 mwei'
BumpPercent = 40
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 12
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 2000
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
```

DeathDeclarationDelay = '10s'

[OCR]

ContractConfirmations = 1

ContractTransmitterTransmitTimeout = '10s'

DatabaseTimeout = '10s'

DeltaCOverride = '168h0m0s'

DeltaCJitterOverride = '1h0m0s'

ObservationGracePeriod = '1s'

[OCR2]

[OCR2.Automation]

GasLimit = 5400000

Kroma Sepolia (2358)

toml

AutoCreateKey = true

BlockBackfillDepth = 10

BlockBackfillSkip = false

ChainType = 'kroma'

FinalityDepth = 400

FinalityTagEnabled = false

LogBackfillBatchSize = 1000

LogPollInterval = '2s'

LogKeepBlocksDepth = 100000

LogPrunePageSize = 0

BackupLogPollerBlockDelay = 100

MinIncomingConfirmations = 1

MinContractPayment = '0.00001 link'

NonceAutoSync = true

NoNewHeadsThreshold = '40s'

RPCDefaultBatchSize = 250

RPCBlockQueryDelay = 1

FinalizedBlockOffset = 0

[Transactions]

ForwardersEnabled = false

MaxInFlight = 16

MaxQueued = 250

ReaperInterval = '1h0m0s'

ReaperThreshold = '168h0m0s'

ResendAfterThreshold = '30s'

[Transactions.AutoPurge]

Enabled = false

[BalanceMonitor]

Enabled = true

[GasEstimator]

Mode = 'BlockHistory'

PriceDefault = '20 gwei'

PriceMax =

'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'

PriceMin = '1 wei'

LimitDefault = 500000

LimitMax = 500000

LimitMultiplier = '1'

LimitTransfer = 21000

BumpMin = '100 wei'

BumpPercent = 20

```

BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 400
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 5400000

```

Polygon Zkevm Cardona (2442)

```

toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'zkevm'
FinalityDepth = 500
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '30s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '12m0s'
RPCDefaultBatchSize = 100
RPCBlockQueryDelay = 1

```

FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '3m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '20 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '1 mwei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '20 mwei'  
BumpPercent = 40  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 12  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 2000  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'

```
DeltaCOVERRIDE = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Fantom Testnet (4002)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0xfaFedb041c0DD4fA2Dc0d87a6B0979Ee6FA7af5F'
LogBackfillBatchSize = 1000
LogPollInterval = '1s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 2
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'SuggestedPrice'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 3800000
```

Klaytn Mainnet (8217)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
```

ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'SuggestedPrice'  
PriceDefault = '750 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '1 gwei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 5  
EIP1559DynamicFees = false  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 8  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Base Mainnet (8453)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'optimismBedrock'
FinalityDepth = 200
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '2s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '40s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '30s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 wei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '100 wei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```



```
[HeadTracker]
HistoryDepth = 300
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 6500000
```

Gnosis Chiado (10200)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'gnosis'
FinalityDepth = 100
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '5s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '20 gwei'  
PriceMax = '500 gwei'  
PriceMin = '1 gwei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = true  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 8  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 4  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Arbitrum Mainnet (42161)

toml  
AutoCreateKey = true  
BlockBackfillDepth = 10  
BlockBackfillSkip = false

```
ChainType = 'arbitrum'
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0xf97f4df75117a78c1A5a0DBb814Af92458539FB4'
LogBackfillBatchSize = 1000
LogPollInterval = '1s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'Arbitrum'
PriceDefault = '100 mwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '0'
LimitDefault = 500000
LimitMax = 10000000000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 5
EIP1559DynamicFees = false
FeeCapDefault = '1 micro'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 0
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 14500000
```

Celo Mainnet (42220)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'celo'
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '5s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '1m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '5 gwei'
PriceMax = '500 gwei'
```

```
PriceMin = '5 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '2 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 12
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 50
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Avalanche Fuji (43113)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 10
FinalityTagEnabled = false
LinkContractAddress = '0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846'
LogBackfillBatchSize = 1000
LogPollInterval = '3s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
```

BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 1  
MinContractPayment = '0.00001 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '30s'  
RPCDefaultBatchSize = 250  
RPCBlockQueryDelay = 2  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '25 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '25 gwei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 24  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = false

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'

```
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Avalanche Mainnet (43114)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 10
FinalityTagEnabled = false
LinkContractAddress = '0x5947BB275c521040051D82396192181b413227A3'
LogBackfillBatchSize = 1000
LogPollInterval = '3s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 2
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '25 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '25 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
```

```
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Celo Testnet (44787)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'celo'
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '5s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '1m0s'
RPCDefaultBatchSize = 250
```



```
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0

[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]
Enabled = false

[BalanceMonitor]
Enabled = true

[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '5 gwei'
PriceMax = '500 gwei'
PriceMin = '5 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '2 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 50
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
```

```
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Linea Goerli (59140)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 15
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '3m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 40
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
```

BlockHistorySize = 8  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 4  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Linea Sepolia (59141)

toml  
AutoCreateKey = true  
BlockBackfillDepth = 10  
BlockBackfillSkip = false  
FinalityDepth = 900  
FinalityTagEnabled = false  
LogBackfillBatchSize = 1000  
LogPollInterval = '15s'  
LogKeepBlocksDepth = 100000  
LogPrunePageSize = 0  
BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 3  
MinContractPayment = '0.00001 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '0s'  
RPCDefaultBatchSize = 250  
RPCBlockQueryDelay = 1  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '3m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '20 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '1 wei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = true  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 8  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 1000  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 4  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Linea Mainnet (59144)

toml

AutoCreateKey = true  
BlockBackfillDepth = 10  
BlockBackfillSkip = false  
FinalityDepth = 300  
FinalityTagEnabled = false  
LogBackfillBatchSize = 1000  
LogPollInterval = '15s'  
LogKeepBlocksDepth = 100000  
LogPrunePageSize = 0  
BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 3  
MinContractPayment = '0.00001 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '0s'  
RPCDefaultBatchSize = 250  
RPCBlockQueryDelay = 1  
FinalizedBlockOffset = 0

[Transactions]

ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '3m0s'

[Transactions.AutoPurge]

Enabled = false

[BalanceMonitor]

Enabled = true

[GasEstimator]

Mode = 'BlockHistory'  
PriceDefault = '20 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '400 mwei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 40  
BumpThreshold = 3  
EIP1559DynamicFees = false  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]

BatchSize = 25  
BlockHistorySize = 8  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]

HistoryDepth = 350  
MaxBufferSize = 3

```
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Metis Sepolia (59902)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'metis'
FinalityDepth = 10
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'SuggestedPrice'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '0'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 0
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Polygon Mumbai (80001)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 500
```

```
FinalityTagEnabled = false
LinkContractAddress = '0x326C977E6efc84E512bB9C30f76E30c160eD06FB'
LogBackfillBatchSize = 1000
LogPollInterval = '1s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 5
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 100
RPCBlockQueryDelay = 10
FinalizedBlockOffset = 0

[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 5000
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]
Enabled = false

[BalanceMonitor]
Enabled = true

[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '1 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '20 gwei'
BumpPercent = 20
BumpThreshold = 5
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 2000
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
```



```
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Polygon Amoy (80002)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 500
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '1s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 5
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 100
RPCBlockQueryDelay = 10
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 5000
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
```

```
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '20 gwei'
BumpPercent = 20
BumpThreshold = 5
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 2000
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Base Goerli (84531)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'optimismBedrock'
FinalityDepth = 200
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '2s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
```

```
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '40s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '30s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 wei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '100 wei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 60
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 300
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
```

DeathDeclarationDelay = '10s'

[OCR]

ContractConfirmations = 1

ContractTransmitterTransmitTimeout = '10s'

DatabaseTimeout = '10s'

DeltaCOverride = '168h0m0s'

DeltaCJitterOverride = '1h0m0s'

ObservationGracePeriod = '1s'

[OCR2]

[OCR2.Automation]

GasLimit = 6500000

Base Sepolia (84532)

toml

AutoCreateKey = true

BlockBackfillDepth = 10

BlockBackfillSkip = false

ChainType = 'optimismBedrock'

FinalityDepth = 200

FinalityTagEnabled = false

LogBackfillBatchSize = 1000

LogPollInterval = '2s'

LogKeepBlocksDepth = 100000

LogPrunePageSize = 0

BackupLogPollerBlockDelay = 100

MinIncomingConfirmations = 1

MinContractPayment = '0.00001 link'

NonceAutoSync = true

NoNewHeadsThreshold = '40s'

RPCDefaultBatchSize = 250

RPCBlockQueryDelay = 1

FinalizedBlockOffset = 0

[Transactions]

ForwardersEnabled = false

MaxInFlight = 16

MaxQueued = 250

ReaperInterval = '1h0m0s'

ReaperThreshold = '168h0m0s'

ResendAfterThreshold = '30s'

[Transactions.AutoPurge]

Enabled = false

[BalanceMonitor]

Enabled = true

[GasEstimator]

Mode = 'BlockHistory'

PriceDefault = '20 gwei'

PriceMax =

'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'

PriceMin = '1 wei'

LimitDefault = 500000

LimitMax = 500000

LimitMultiplier = '1'

LimitTransfer = 21000

BumpMin = '100 wei'

BumpPercent = 20

```
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 60
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 300
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 6500000
```

Arbitrum Rinkeby (421611)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'arbitrum'
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x615fBe6372676474d9e6933d310469c9b68e9726'
LogBackfillBatchSize = 1000
LogPollInterval = '1s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
```

RPCBlockQueryDelay = 1  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'Arbitrum'  
PriceDefault = '100 mwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '0'  
LimitDefault = 500000  
LimitMax = 10000000000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 5  
EIP1559DynamicFees = false  
FeeCapDefault = '1 micro'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 0  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 10  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'

```
DatabaseTimeout = '10s'
DeltaCOVERRIDE = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Arbitrum Goerli (421613)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'arbitrum'
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0xd14838A68E8AFBAdE5efb411d5871ea0011AFd28'
LogBackfillBatchSize = 1000
LogPollInterval = '1s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'Arbitrum'
PriceDefault = '100 mwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '0'
LimitDefault = 500000
LimitMax = 10000000000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 5
EIP1559DynamicFees = false
FeeCapDefault = '1 micro'
TipCapDefault = '1 wei'
```

TipCapMin = '1 wei'

[GasEstimator.BlockHistory]

BatchSize = 25

BlockHistorySize = 0

CheckInclusionBlocks = 12

CheckInclusionPercentile = 90

TransactionPercentile = 60

[HeadTracker]

HistoryDepth = 100

MaxBufferSize = 3

SamplingInterval = '1s'

MaxAllowedFinalityDepth = 10000

FinalityTagBypass = true

[NodePool]

PollFailureThreshold = 5

PollInterval = '10s'

SelectionMode = 'HighestHead'

SyncThreshold = 10

LeaseDuration = '0s'

NodeIsSyncingEnabled = false

FinalizedBlockPollInterval = '5s'

EnforceRepeatableRead = false

DeathDeclarationDelay = '10s'

[OCR]

ContractConfirmations = 1

ContractTransmitterTransmitTimeout = '10s'

DatabaseTimeout = '10s'

DeltaCOverride = '168h0m0s'

DeltaCJitterOverride = '1h0m0s'

ObservationGracePeriod = '1s'

[OCR2]

[OCR2.Automation]

GasLimit = 14500000

Arbitrum Sepolia (421614)

toml

AutoCreateKey = true

BlockBackfillDepth = 10

BlockBackfillSkip = false

ChainType = 'arbitrum'

FinalityDepth = 50

FinalityTagEnabled = false

LogBackfillBatchSize = 1000

LogPollInterval = '1s'

LogKeepBlocksDepth = 100000

LogPrunePageSize = 0

BackupLogPollerBlockDelay = 100

MinIncomingConfirmations = 3

MinContractPayment = '0.00001 link'

NonceAutoSync = true

NoNewHeadsThreshold = '0s'

RPCDefaultBatchSize = 250

RPCBlockQueryDelay = 1

FinalizedBlockOffset = 0

[Transactions]

ForwardersEnabled = false



MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'Arbitrum'  
PriceDefault = '100 mwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '0'  
LimitDefault = 500000  
LimitMax = 10000000000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '5 gwei'  
BumpPercent = 20  
BumpThreshold = 5  
EIP1559DynamicFees = false  
FeeCapDefault = '1 micro'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 0  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 100  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 10  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

```
[OCR2]
[OCR2.Automation]
GasLimit = 14500000
```

Scroll Sepolia (534351)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'scroll'
FinalityDepth = 10
FinalityTagEnabled = true
LogBackfillBatchSize = 1000
LogPollInterval = '5s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 wei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '1 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 24
CheckInclusionBlocks = 12
```

CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 50  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Scroll Mainnet (534352)

toml  
AutoCreateKey = true  
BlockBackfillDepth = 10  
BlockBackfillSkip = false  
ChainType = 'scroll'  
FinalityDepth = 10  
FinalityTagEnabled = true  
LogBackfillBatchSize = 1000  
LogPollInterval = '5s'  
LogKeepBlocksDepth = 100000  
LogPrunePageSize = 0  
BackupLogPollerBlockDelay = 100  
MinIncomingConfirmations = 1  
MinContractPayment = '0.00001 link'  
NonceAutoSync = true  
NoNewHeadsThreshold = '0s'  
RPCDefaultBatchSize = 250  
RPCBlockQueryDelay = 1  
FinalizedBlockOffset = 0

[Transactions]  
ForwardersEnabled = false  
MaxInFlight = 16  
MaxQueued = 250  
ReaperInterval = '1h0m0s'  
ReaperThreshold = '168h0m0s'  
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]  
Enabled = false

[BalanceMonitor]  
Enabled = true

[GasEstimator]  
Mode = 'BlockHistory'  
PriceDefault = '20 gwei'  
PriceMax =  
'115792089237316195423570985008687907853269984665.640564039457584007913129639935  
tether'  
PriceMin = '1 wei'  
LimitDefault = 500000  
LimitMax = 500000  
LimitMultiplier = '1'  
LimitTransfer = 21000  
BumpMin = '1 gwei'  
BumpPercent = 20  
BumpThreshold = 3  
EIP1559DynamicFees = true  
FeeCapDefault = '100 gwei'  
TipCapDefault = '1 wei'  
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]  
BatchSize = 25  
BlockHistorySize = 24  
CheckInclusionBlocks = 12  
CheckInclusionPercentile = 90  
TransactionPercentile = 60

[HeadTracker]  
HistoryDepth = 50  
MaxBufferSize = 3  
SamplingInterval = '1s'  
MaxAllowedFinalityDepth = 10000  
FinalityTagBypass = true

[NodePool]  
PollFailureThreshold = 5  
PollInterval = '10s'  
SelectionMode = 'HighestHead'  
SyncThreshold = 5  
LeaseDuration = '0s'  
NodeIsSyncingEnabled = false  
FinalizedBlockPollInterval = '5s'  
EnforceRepeatableRead = false  
DeathDeclarationDelay = '10s'

[OCR]  
ContractConfirmations = 1  
ContractTransmitterTransmitTimeout = '10s'  
DatabaseTimeout = '10s'  
DeltaCOverride = '168h0m0s'  
DeltaCJitterOverride = '1h0m0s'  
ObservationGracePeriod = '1s'

[OCR2]  
[OCR2.Automation]  
GasLimit = 5400000

Ethereum Sepolia (11155111)

```

toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x779877A7B0D9E8603169DdbD7836e478b4624789'
LogBackfillBatchSize = 1000
LogPollInterval = '15s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 3
MinContractPayment = '0.1 link'
NonceAutoSync = true
NoNewHeadsThreshold = '3m0s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0

[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]
Enabled = false

[BalanceMonitor]
Enabled = true

[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 4
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 50

[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3

```

```
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = false
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 10500000
```

Optimism Sepolia (11155420)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
ChainType = 'optimismBedrock'
FinalityDepth = 200
FinalityTagEnabled = false
LogBackfillBatchSize = 1000
LogPollInterval = '2s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '40s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '30s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '20 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 wei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '100 wei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = true
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'
```

```
[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 60
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60
```

```
[HeadTracker]
HistoryDepth = 300
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true
```

```
[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 10
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 1
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 65000000
```

Harmony Mainnet (1666600000)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
```

```
FinalityTagEnabled = false
LinkContractAddress = '0x218532a12a389a4a92fC0C5Fb22901D1c19198aA'
LogBackfillBatchSize = 1000
LogPollInterval = '2s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0

[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'

[Transactions.AutoPurge]
Enabled = false

[BalanceMonitor]
Enabled = true

[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '5 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
```



```
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'
```

```
[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'
```

```
[OCR2]
[OCR2.Automation]
GasLimit = 5400000
```

Harmony Testnet (1666700000)

```
toml
AutoCreateKey = true
BlockBackfillDepth = 10
BlockBackfillSkip = false
FinalityDepth = 50
FinalityTagEnabled = false
LinkContractAddress = '0x8b12Ac23BFe11cAb03a634C1F117D64a7f2cFD3e'
LogBackfillBatchSize = 1000
LogPollInterval = '2s'
LogKeepBlocksDepth = 100000
LogPrunePageSize = 0
BackupLogPollerBlockDelay = 100
MinIncomingConfirmations = 1
MinContractPayment = '0.00001 link'
NonceAutoSync = true
NoNewHeadsThreshold = '30s'
RPCDefaultBatchSize = 250
RPCBlockQueryDelay = 1
FinalizedBlockOffset = 0
```

```
[Transactions]
ForwardersEnabled = false
MaxInFlight = 16
MaxQueued = 250
ReaperInterval = '1h0m0s'
ReaperThreshold = '168h0m0s'
ResendAfterThreshold = '1m0s'
```

```
[Transactions.AutoPurge]
Enabled = false
```

```
[BalanceMonitor]
Enabled = true
```

```
[GasEstimator]
Mode = 'BlockHistory'
PriceDefault = '5 gwei'
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether'
```

```

PriceMin = '1 gwei'
LimitDefault = 500000
LimitMax = 500000
LimitMultiplier = '1'
LimitTransfer = 21000
BumpMin = '5 gwei'
BumpPercent = 20
BumpThreshold = 3
EIP1559DynamicFees = false
FeeCapDefault = '100 gwei'
TipCapDefault = '1 wei'
TipCapMin = '1 wei'

[GasEstimator.BlockHistory]
BatchSize = 25
BlockHistorySize = 8
CheckInclusionBlocks = 12
CheckInclusionPercentile = 90
TransactionPercentile = 60

[HeadTracker]
HistoryDepth = 100
MaxBufferSize = 3
SamplingInterval = '1s'
MaxAllowedFinalityDepth = 10000
FinalityTagBypass = true

[NodePool]
PollFailureThreshold = 5
PollInterval = '10s'
SelectionMode = 'HighestHead'
SyncThreshold = 5
LeaseDuration = '0s'
NodeIsSyncingEnabled = false
FinalizedBlockPollInterval = '5s'
EnforceRepeatableRead = false
DeathDeclarationDelay = '10s'

[OCR]
ContractConfirmations = 4
ContractTransmitterTransmitTimeout = '10s'
DatabaseTimeout = '10s'
DeltaCOverride = '168h0m0s'
DeltaCJitterOverride = '1h0m0s'
ObservationGracePeriod = '1s'

[OCR2]
[OCR2.Automation]
GasLimit = 5400000

```

ChainID

```

toml
ChainID = '1' Example

```

ChainID is the EVM chain ID. Mandatory.

Enabled

```

toml
Enabled = true Default

```

Enabled enables this chain.

AutoCreateKey

```
toml
AutoCreateKey = true Default
```

AutoCreateKey, if set to true, will ensure that there is always at least one transmit key for the given chain.

BlockBackfillDepth

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
BlockBackfillDepth = 10 Default
```

BlockBackfillDepth specifies the number of blocks before the current HEAD that the log broadcaster will try to re-consume logs from.

BlockBackfillSkip

```
toml
BlockBackfillSkip = false Default
```

BlockBackfillSkip enables skipping of very long backfills.

ChainType

```
toml
ChainType = 'arbitrum' Example
```

ChainType is automatically detected from chain ID. Set this to force a certain chain type regardless of chain ID.  
Available types: arbitrum, celo, gnosis, kroma, metis, optimismBedrock, scroll, wemix, xlayer, zksync

FinalityDepth

```
toml
FinalityDepth = 50 Default
```

FinalityDepth is the number of blocks after which an ethereum transaction is considered "final". Note that the default is automatically set based on chain ID, so it should not be necessary to change this under normal operation. BlocksConsideredFinal determines how deeply we look back to ensure that transactions are confirmed onto the longest chain  
There is not a large performance penalty to setting this relatively high (on the order of hundreds)  
It is practically limited by the number of heads we store in the database and should be less than this with a comfortable margin.  
If a transaction is mined in a block more than this many blocks ago, and is reorged out, we will NOT retransmit this transaction and undefined behaviour can occur including gaps in the nonce sequence that require manual intervention to fix.  
Therefore, this number represents a number of blocks we consider large enough that no re-org this deep will ever feasibly happen.

Special cases:

FinalityDepth=0 would imply that transactions can be final even before they were mined into a block. This is not supported.

FinalityDepth=1 implies that transactions are final after we see them in one block.

Examples:

Transaction sending:

A transaction is sent at block height 42

FinalityDepth is set to 5

A re-org occurs at height 44 starting at block 41, transaction is marked for rebroadcast

A re-org occurs at height 46 starting at block 41, transaction is marked for rebroadcast

A re-org occurs at height 47 starting at block 41, transaction is NOT marked for rebroadcast

FinalityTagEnabled

toml

FinalityTagEnabled = false Default

FinalityTagEnabled means that the chain supports the finalized block tag when querying for a block. If FinalityTagEnabled is set to true for a chain, then FinalityDepth field is ignored.

Finality for a block is solely defined by the finality related tags provided by the chain's RPC API. This is a placeholder and hasn't been implemented yet.

FlagsContractAddress

⚠️, ADVANCED: Do not change this setting unless you know what you are doing.

toml

FlagsContractAddress = '0xae4E781a6218A8031764928E88d457937A954fC3' Example

FlagsContractAddress can optionally point to a Flags contract. If set, the node will lookup that contract for each job that supports flags contracts (currently OCR and FM jobs are supported). If the job's contractAddress is set as hibernating in the FlagsContractAddress address, it overrides the standard update parameters (such as heartbeat/threshold).

LinkContractAddress

toml

LinkContractAddress = '0x538aAaB4ea120b2bC2fe5D296852D948F07D849e' Example

LinkContractAddress is the canonical ERC-677 LINK token contract address on the given chain. Note that this is usually autodetected from chain ID.

LogBackfillBatchSize

⚠️, ADVANCED: Do not change this setting unless you know what you are doing.

toml

LogBackfillBatchSize = 1000 Default

LogBackfillBatchSize sets the batch size for calling FilterLogs when we backfill missing logs.

## LogPollInterval

â§ i, **ADVANCED:** Do not change this setting unless you know what you are doing.

toml

LogPollInterval = '15s' Default

`LogPollInterval` works in conjunction with `Feature.LogPoller`. Controls how frequently the log poller polls for logs. Defaults to the block production rate.

## LogKeepBlocksDepth

â§ ĩ, ADVANCED: Do not change this setting unless you know what you are doing.

toml

LogKeepBlocksDepth = 100000 Default

`LogKeepBlocksDepth` works in conjunction with `Feature.LogPoller`. Controls how many blocks the poller will keep, must be greater than `FinalityDepth+1`.

## LogPrunePageSize

â¸ i, ADVANCED: Do not change this setting unless you know what you are doing.

toml

LogPrunePageSize = 0 Default

LogPrunePageSize defines size of the page for pruning logs. Controls how many logs/blocks (at most) are deleted in a single prune tick. Default value 0 means no paging, delete everything at once.

## BackupLogPollerBlockDelay

â§ i. **ADVANCED:** Do not change this setting unless you know what you are doing.

toml

```
BackupLogPollerBlockDelay = 100 Default
```

BackupLogPollerBlockDelay works in conjunction with Feature.LogPoller. Controls the block delay of Backup LogPoller, affecting how far behind the latest finalized block it starts and how often it runs.

BackupLogPollerDelay=0 will disable Backup LogPoller (not recommended for production environment).

## MinContractPayment

toml

```
MinContractPayment = '100000000000000 juels' Default
```

MinContractPayment is the minimum payment in LINK required to execute a direct request job. This can be overridden on a per-job basis.

## MinIncomingConfirmations

toml

```
MinIncomingConfirmations = 3 Default
```

MinIncomingConfirmations is the minimum required confirmations before a log event will be consumed.

#### NonceAutoSync

```
toml
NonceAutoSync = true Default
```

NonceAutoSync enables automatic nonce syncing on startup. Chainlink nodes will automatically try to sync its local nonce with the remote chain on startup and fast forward if necessary. This is almost always safe but can be disabled in exceptional cases by setting this value to false.

#### NoNewHeadsThreshold

```
toml
NoNewHeadsThreshold = '3m' Default
```

NoNewHeadsThreshold controls how long to wait after receiving no new heads before NodePool marks rpc endpoints as out-of-sync, and HeadTracker logs warnings.

Set to zero to disable out-of-sync checking.

#### OperatorFactoryAddress

```
toml
OperatorFactoryAddress = '0xa5B85635Be42F21f94F28034B7DA440EeFF0F418' Example
```

OperatorFactoryAddress is the address of the canonical operator forwarder contract on the given chain. Note that this is usually autodetected from chain ID.

#### RPCDefaultBatchSize

```
toml
RPCDefaultBatchSize = 250 Default
```

RPCDefaultBatchSize is the default batch size for batched RPC calls.

#### RPCBlockQueryDelay

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
RPCBlockQueryDelay = 1 Default
```

RPCBlockQueryDelay controls the number of blocks to trail behind head in the block history estimator and balance monitor.  
For example, if this is set to 3, and we receive block 10, block history estimator will fetch block 7.

CAUTION: You might be tempted to set this to 0 to use the latest possible block, but it is possible to receive a head BEFORE that block is actually available from the connected node via RPC, due to race conditions in the code of the remote ETH node. In this case you will get false "zero" blocks that are missing transactions.

#### FinalizedBlockOffset

```
toml
FinalizedBlockOffset = 0 Default
```

FinalizedBlockOffset defines the number of blocks by which the latest finalized block will be shifted/delayed.

For example, suppose RPC returns block 100 as the latest finalized. In that case, the CL Node will treat block 100 - FinalizedBlockOffset as the latest finalized block and latest - FinalityDepth - FinalizedBlockOffset in case of FinalityTagEnabled = false.

With EnforceRepeatableRead = true, RPC is considered healthy only if its most recent finalized block is larger or equal to the highest finalized block observed by the CL Node minus FinalizedBlockOffset.

Higher values of FinalizedBlockOffset with EnforceRepeatableRead = true reduce the number of false FinalizedBlockOutOfSync declarations on healthy RPCs that are slightly lagging behind due to network delays.

This may increase the number of healthy RPCs and reduce the probability that the CL Node will not have any healthy alternatives to the active RPC.

CAUTION: Setting this to values higher than 0 may delay transaction creation in products (e.g., CCIP, Automation) that base their decision on finalized on-chain events.

PoS chains with FinalityTagEnabled=true and batched (epochs) blocks finalization (e.g., Ethereum Mainnet) must be treated with special care as a minor increase in the FinalizedBlockOffset may lead to significant delays.

For example, let's say that FinalizedBlockOffset = 1 and blocks are finalized in batches of 32.

The latest finalized block on chain is 64, so block 63 is the latest finalized for CL Node.

Block 64 will be treated as finalized by CL Node only when chain's latest finalized block is 65. As chain finalizes blocks in batches of 32, CL Node has to wait for a whole new batch to be finalized to treat block 64 as finalized.

## EVM.Transactions

```
toml
[EVM.Transactions]
ForwardersEnabled = false Default
MaxInFlight = 16 Default
MaxQueued = 250 Default
ReaperInterval = '1h' Default
ReaperThreshold = '168h' Default
ResendAfterThreshold = '1m' Default
```

### ForwardersEnabled

```
toml
ForwardersEnabled = false Default
```

ForwardersEnabled enables or disables sending transactions through forwarder contracts.

### MaxInFlight

```
toml
MaxInFlight = 16 Default
```

MaxInFlight controls how many transactions are allowed to be "in-flight" i.e. broadcast but unconfirmed at any one time. You can consider this a form of transaction throttling.

The default is set conservatively at 16 because this is a pessimistic minimum that both geth and parity will hold without evicting local transactions. If your node is falling behind and you need higher throughput, you can increase this setting, but you MUST make sure that your ETH node is configured properly otherwise you can get nonce gapped and your node will get stuck.

0 value disables the limit. Use with caution.

#### MaxQueued

```
toml
MaxQueued = 250 Default
```

MaxQueued is the maximum number of unbroadcast transactions per key that are allowed to be enqueued before jobs will start failing and rejecting send of any further transactions. This represents a sanity limit and generally indicates a problem with your ETH node (transactions are not getting mined).

Do NOT blindly increase this value thinking it will fix things if you start hitting this limit because transactions are not getting mined, you will instead only make things worse.

In deployments with very high burst rates, or on chains with large re-orgs, you may consider increasing this.

0 value disables any limit on queue size. Use with caution.

#### ReaperInterval

```
toml
ReaperInterval = '1h' Default
```

ReaperInterval controls how often the EthTx reaper will run.

#### ReaperThreshold

```
toml
ReaperThreshold = '168h' Default
```

ReaperThreshold indicates how old an EthTx ought to be before it can be reaped.

#### ResendAfterThreshold

```
toml
ResendAfterThreshold = '1m' Default
```

ResendAfterThreshold controls how long to wait before re-broadcasting a transaction that has not yet been confirmed.

#### EVM.Transactions.AutoPurge

```
toml
[EVM.Transactions.AutoPurge]
Enabled = false Default
DetectionApiUrl = 'https://example.api.io' Example
Threshold = 5 Example
MinAttempts = 3 Example
```



Enabled

```
toml
Enabled = false Default
```

Enabled enables or disables automatically purging transactions that have been identified as terminally stuck (will never be included on-chain). This feature is only expected to be used by ZK chains.

DetectionApiUrl

```
toml
DetectionApiUrl = 'https://example.api.io' Example
```

DetectionApiUrl configures the base url of a custom endpoint used to identify terminally stuck transactions.

Threshold

```
toml
Threshold = 5 Example
```

Threshold configures the number of blocks a transaction has to remain unconfirmed before it is evaluated for being terminally stuck. This threshold is only applied if there is no custom API to identify stuck transactions provided by the chain.

MinAttempts

```
toml
MinAttempts = 3 Example
```

MinAttempts configures the minimum number of broadcasted attempts a transaction has to have before it is evaluated further for being terminally stuck. This threshold is only applied if there is no custom API to identify stuck transactions provided by the chain. Ensure the gas estimator configs take more bump attempts before reaching the configured max gas price.

EVM.BalanceMonitor

```
toml
[EVM.BalanceMonitor]
Enabled = true Default
```

Enabled

```
toml
Enabled = true Default
```

Enabled balance monitoring for all keys.

EVM.GasEstimator

```
toml
[EVM.GasEstimator]
Mode = 'BlockHistory' Default
PriceDefault = '20 gwei' Default
PriceMax =
```

```
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether' Default
PriceMin = '1 gwei' Default
LimitDefault = 500000 Default
LimitMax = 500000 Default
LimitMultiplier = '1.0' Default
LimitTransfer = 21000 Default
BumpMin = '5 gwei' Default
BumpPercent = 20 Default
BumpThreshold = 3 Default
BumpTxDepth = 16 Example
EIP1559DynamicFees = false Default
FeeCapDefault = '100 gwei' Default
TipCapDefault = '1 wei' Default
TipCapMin = '1 wei' Default
```

Mode

```
toml
Mode = 'BlockHistory' Default
```

Mode controls what type of gas estimator is used.

- FixedPrice uses static configured values for gas price (can be set via API call).
- BlockHistory dynamically adjusts default gas price based on heuristics from mined blocks.
- L2Suggested mode is deprecated and replaced with SuggestedPrice.
- SuggestedPrice is a mode which uses the gas price suggested by the rpc endpoint via ethgasPrice.
- Arbitrum is a special mode only for use with Arbitrum blockchains. It uses the suggested gas price (up to ETHMAXGASPRICEWEI, with 1000 gwei default) as well as an estimated gas limit (up to ETHGASLIMITMAX, with 1,000,000,000 default).

Chainlink nodes decide what gas price to use using an Estimator. It ships with several simple and battle-hardened built-in estimators that should work well for almost all use-cases. Note that estimators will change their behaviour slightly depending on if you are in EIP-1559 mode or not.

You can also use your own estimator for gas price by selecting the FixedPrice estimator and using the exposed API to set the price.

An important point to note is that the Chainlink node does not ship with built-in support for go-ethereum's estimateGas call. This is for several reasons, including security and reliability. We have found empirically that it is not generally safe to rely on the remote ETH node's idea of what gas price should be.

PriceDefault

```
toml
PriceDefault = '20 gwei' Default
```

PriceDefault is the default gas price to use when submitting transactions to the blockchain. Will be overridden by the built-in BlockHistoryEstimator if enabled, and might be increased if gas bumping is enabled.

(Only applies to legacy transactions)

Can be used with the chainlink setgasprice to be updated while the node is still running.

## PriceMax

```
toml
PriceMax =
'115792089237316195423570985008687907853269984665.640564039457584007913129639935
tether' Default
```

PriceMax is the maximum gas price. Chainlink nodes will never pay more than this for a transaction.

This applies to both legacy and EIP1559 transactions.

Note that it is impossible to disable the maximum limit. Setting this value to zero will prevent paying anything for any transaction (which can be useful in some rare cases).

Most chains by default have the maximum set to  $2^{256}-1$  Wei which is the maximum allowed gas price on EVM-compatible chains, and is so large it may as well be unlimited.

## PriceMin

```
toml
PriceMin = '1 gwei' Default
```

PriceMin is the minimum gas price. Chainlink nodes will never pay less than this for a transaction.

(Only applies to legacy transactions)

It is possible to force the Chainlink node to use a fixed gas price by setting a combination of these, e.g.

```
toml
EIP1559DynamicFees = false
PriceMax = 100
PriceMin = 100
PriceDefault = 100
BumpThreshold = 0
Mode = 'FixedPrice'
```

## LimitDefault

```
toml
LimitDefault = 500000 Default
```

LimitDefault sets default gas limit for outgoing transactions. This should not need to be changed in most cases.

Some job types, such as Keeper jobs, might set their own gas limit unrelated to this value.

## LimitMax

```
toml
LimitMax = 500000 Default
```

LimitMax sets a maximum for estimated gas limits. This currently only applies to Arbitrum GasEstimatorMode.

## LimitMultiplier

```
toml
LimitMultiplier = '1.0' Default
```

LimitMultiplier is the factor by which a transaction's GasLimit is multiplied before transmission. So if the value is 1.1, and the GasLimit for a transaction is 10, 10% will be added before transmission.

This factor is always applied, so includes L2 transactions which uses a default gas limit of 1 and is also applied to LimitDefault.

LimitTransfer

```
toml
LimitTransfer = 21000 Default
```

LimitTransfer is the gas limit used for an ordinary ETH transfer.

BumpMin

```
toml
BumpMin = '5 gwei' Default
```

BumpMin is the minimum fixed amount of wei by which gas is bumped on each transaction attempt.

BumpPercent

```
toml
BumpPercent = 20 Default
```

BumpPercent is the percentage by which to bump gas on a transaction that has exceeded BumpThreshold. The larger of BumpPercent and BumpMin is taken for gas bumps.

The SuggestedPriceEstimator adds the larger of BumpPercent and BumpMin on top of the price provided by the RPC when bumping a transaction's gas.

BumpThreshold

```
toml
BumpThreshold = 3 Default
```

BumpThreshold is the number of blocks to wait for a transaction stuck in the mempool before automatically bumping the gas price. Set to 0 to disable gas bumping completely.

BumpTxDepth

```
toml
BumpTxDepth = 16 Example
```

BumpTxDepth is the number of transactions to gas bump starting from oldest. Set to 0 for no limit (i.e. bump all). Can not be greater than EVM.Transactions.MaxInFlight. If not set, defaults to EVM.Transactions.MaxInFlight.

EIP1559DynamicFees

toml  
EIP1559DynamicFees = false Default

EIP1559DynamicFees forces EIP-1559 transaction mode. Enabling EIP-1559 mode can help reduce gas costs on chains that support it. This is supported only on official Ethereum mainnet and testnets. It is not recommended to enable this setting on Polygon because the EIP-1559 fee market appears to be broken on all Polygon chains and EIP-1559 transactions are less likely to be included than legacy transactions.

#### Technical details

Chainlink nodes include experimental support for submitting transactions using type 0x2 (EIP-1559) envelope.

EIP-1559 mode is enabled by default on the Ethereum Mainnet, but can be enabled on a per-chain basis or globally.

This might help to save gas on spikes. Chainlink nodes should react faster on the upleg and avoid overpaying on the downleg. It might also be possible to set `EVM.GasEstimator.BlockHistory.BatchSize` to a smaller value such as 12 or even 6 because tip cap should be a more consistent indicator of inclusion time than total gas price. This would make Chainlink nodes more responsive and should reduce response time variance. Some experimentation is required to find optimum settings.

Set with caution, if you set this on a chain that does not actually support EIP-1559 your node will be broken.

In EIP-1559 mode, the total price for the transaction is the minimum of base fee + tip cap and fee cap. More information can be found on the official EIP.

Chainlink's implementation of EIP-1559 works as follows:

If you are using `FixedPriceEstimator`:

- With gas bumping disabled, it will submit all transactions with `feecap=PriceMax` and `tipcap=GasTipCapDefault`
- With gas bumping enabled, it will submit all transactions initially with `feecap=GasFeeCapDefault` and `tipcap=GasTipCapDefault`.

If you are using `BlockHistoryEstimator` (default for most chains):

- With gas bumping disabled, it will submit all transactions with `feecap=PriceMax` and `tipcap=<calculated using past blocks>`
- With gas bumping enabled (default for most chains) it will submit all transactions initially with `feecap = ( current block base fee (1.125 ^ N) + tipcap )` where N is configurable by setting `EVM.GasEstimator.BlockHistory.EIP1559FeeCapBufferBlocks` but defaults to `gas bump threshold+1` and `tipcap=<calculated using past blocks>`

Bumping works as follows:

- Increase tipcap by `max(tipcap * (1 + BumpPercent), tipcap + BumpMin)`
- Increase feecap by `max(feecap * (1 + BumpPercent), feecap + BumpMin)`

A quick note on terminology - Chainlink nodes use the same terms used internally by go-ethereum source code to describe various prices. This is not the same as the externally used terms. For reference:

- Base Fee Per Gas = `BaseFeePerGas`
- Max Fee Per Gas = `FeeCap`
- Max Priority Fee Per Gas = `TipCap`

In EIP-1559 mode, the following changes occur to how configuration works:

- All new transactions will be sent as type 0x2 transactions specifying a TipCap and FeeCap. Be aware that existing pending legacy transactions will continue to be gas bumped in legacy mode.
- BlockHistoryEstimator will apply its calculations (gas percentile etc) to the TipCap and this value will be used for new transactions (GasPrice will be ignored)
- FixedPriceEstimator will use GasTipCapDefault instead of GasPriceDefault for the tip cap
- FixedPriceEstimator will use GasFeeCapDefault instead of GasPriceDefault for the fee cap
- PriceMin is ignored for new transactions and GasTipCapMinimum is used instead (default 0)
- PriceMax still represents that absolute upper limit that Chainlink will ever spend (total) on a single tx
- Keeper.GasTipCapBufferPercent is ignored in EIP-1559 mode and Keeper.GasTipCapBufferPercent is used instead

FeeCapDefault

```
toml
FeeCapDefault = '100 gwei' Default
```

FeeCapDefault controls the fixed initial fee cap, if EIP1559 mode is enabled and FixedPrice gas estimator is used.

TipCapDefault

```
toml
TipCapDefault = '1 wei' Default
```

TipCapDefault is the default gas tip to use when submitting transactions to the blockchain. Will be overridden by the built-in BlockHistoryEstimator if enabled, and might be increased if gas bumping is enabled.

(Only applies to EIP-1559 transactions)

TipCapMin

```
toml
TipCapMin = '1 wei' Default
```

TipCapMinimum is the minimum gas tip to use when submitting transactions to the blockchain.

(Only applies to EIP-1559 transactions)

EVM.GasEstimator.LimitJobType

```
toml
[EVM.GasEstimator.LimitJobType]
OCR = 100000 Example
OCR2 = 100000 Example
DR = 100000 Example
VRF = 100000 Example
FM = 100000 Example
Keeper = 100000 Example
```

## OCR

toml

OCR = 100000 Example

OCR overrides LimitDefault for OCR jobs.

## OCR2

toml

OCR2 = 100000 Example

OCR2 overrides LimitDefault for OCR2 jobs.

## DR

toml

DR = 100000 Example

DR overrides LimitDefault for Direct Request jobs.

## VRF

toml

VRF = 100000 Example

VRF overrides LimitDefault for VRF jobs.

## FM

toml

FM = 100000 Example

FM overrides LimitDefault for Flux Monitor jobs.

## Keeper

toml

Keeper = 100000 Example

Keeper overrides LimitDefault for Keeper jobs.

## EVM.GasEstimator.BlockHistory

toml

[EVM.GasEstimator.BlockHistory]

BatchSize = 25 Default

BlockHistorySize = 8 Default

CheckInclusionBlocks = 12 Default

CheckInclusionPercentile = 90 Default

EIP1559FeeCapBufferBlocks = 13 Example

TransactionPercentile = 60 Default

These settings allow you to configure how your node calculates gas prices when using the block history estimator.  
In most cases, leaving these values at their defaults should give good results.

## BatchSize

```
toml
BatchSize = 25 Default
```

BatchSize sets the maximum number of blocks to fetch in one batch in the block history estimator.

If the BatchSize variable is set to 0, it defaults to EVM.RPCDefaultBatchSize.

## BlockHistorySize

```
toml
BlockHistorySize = 8 Default
```

BlockHistorySize controls the number of past blocks to keep in memory to use as a basis for calculating a percentile gas price.

## CheckInclusionBlocks

```
toml
CheckInclusionBlocks = 12 Default
```

CheckInclusionBlocks is the number of recent blocks to use to detect if there is a transaction propagation/connectivity issue, and to prevent bumping in these cases.

This can help avoid the situation where RPC nodes are not propagating transactions for some non-price-related reason (e.g. go-ethereum bug, networking issue etc) and bumping gas would not help.

Set to zero to disable connectivity checking completely.

## CheckInclusionPercentile

```
toml
CheckInclusionPercentile = 90 Default
```

CheckInclusionPercentile controls the percentile that a transaction must have been higher than for all the blocks in the inclusion check window in order to register as a connectivity issue.

For example, if CheckInclusionBlocks=12 and CheckInclusionPercentile=90 then further bumping will be prevented for any transaction with any attempt that has a higher price than the 90th percentile for the most recent 12 blocks.

## EIP1559FeeCapBufferBlocks

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
EIP1559FeeCapBufferBlocks = 13 Example
```

EIP1559FeeCapBufferBlocks controls the buffer blocks to add to the current base fee when sending a transaction. By default, the gas bumping threshold + 1 block is used.

(Only applies to EIP-1559 transactions)

## TransactionPercentile



```
toml
TransactionPercentile = 60 Default
```

TransactionPercentile specifies gas price to choose. E.g. if the block history contains four transactions with gas prices [100, 200, 300, 400] then picking 25 for this number will give a value of 200. If the calculated gas price is higher than GasPriceDefault then the higher price will be used as the base price for new transactions.

Must be in range 0-100.

Only has an effect if gas updater is enabled.

Think of this number as an indicator of how aggressive you want your node to price its transactions.

Setting this number higher will cause the Chainlink node to select higher gas prices.

Setting it lower will tend to set lower gas prices.

EVM.HeadTracker

```
toml
[EVM.HeadTracker]
HistoryDepth = 100 Default
MaxBufferSize = 3 Default
SamplingInterval = '1s' Default
FinalityTagBypass = true Default
MaxAllowedFinalityDepth = 10000 Default
```

The head tracker continually listens for new heads from the chain.

In addition to these settings, it log warnings if EVM.NoNewHeadsThreshold is exceeded without any new blocks being emitted.

HistoryDepth

```
toml
HistoryDepth = 100 Default
```

HistoryDepth tracks the top N blocks on top of the latest finalized block to keep in the heads database table.

Note that this can easily result in MORE than N + finality depth records since in the case of re-orgs we keep multiple heads for a particular block height.

This number should be at least as large as FinalityDepth.

There may be a small performance penalty to setting this to something very large (10,000+)

MaxBufferSize

```
toml
MaxBufferSize = 3 Default
```

MaxBufferSize is the maximum number of heads that may be buffered in front of the head tracker before older heads start to be dropped. You may think of it as something like the maximum permissible "lag" for the head tracker before we start dropping heads to keep up.

SamplingInterval

â§ i, ADVANCED: Do not change this setting unless you know what you are doing.

```
toml
SamplingInterval = '1s' Default
```

SamplingInterval means that head tracker callbacks will at maximum be made once in every window of this duration. This is a performance optimisation for fast chains. Set to 0 to disable sampling entirely.

FinalityTagBypass

```
toml
FinalityTagBypass = true Default
```

FinalityTagBypass disables FinalityTag support in HeadTracker and makes it track blocks up to FinalityDepth from the most recent head.

It should only be used on chains with an extremely large actual finality depth (the number of blocks between the most recent head and the latest finalized block).

Has no effect if FinalityTagsEnabled = false

MaxAllowedFinalityDepth

```
toml
MaxAllowedFinalityDepth = 10000 Default
```

MaxAllowedFinalityDepth - defines maximum number of blocks between the most recent head and the latest finalized block.

If actual finality depth exceeds this number, HeadTracker aborts backfill and returns an error.

Has no effect if FinalityTagsEnabled = false

EVM.KeySpecific

```
toml
[[EVM.KeySpecific]]
Key = '0x2a3e23c6f242F5345320814aC8a1b4E58707D292' Example
GasEstimator.PriceMax = '79 gwei' Example
```

Key

```
toml
Key = '0x2a3e23c6f242F5345320814aC8a1b4E58707D292' Example
```

Key is the account to apply these settings to

PriceMax

```
toml
GasEstimator.PriceMax = '79 gwei' Example
```

GasEstimator.PriceMax overrides the maximum gas price for this key. See EVM.GasEstimator.PriceMax.

EVM.NodePool

```
toml
```

```
[EVM.NodePool]
PollFailureThreshold = 5 Default
PollInterval = '10s' Default
SelectionMode = 'HighestHead' Default
SyncThreshold = 5 Default
LeaseDuration = '0s' Default
NodeIsSyncingEnabled = false Default
FinalizedBlockPollInterval = '5s' Default
EnforceRepeatableRead = false Default
DeathDeclarationDelay = '10s' Default
```

The node pool manages multiple RPC endpoints.

In addition to these settings, `EVM.NoNewHeadsThreshold` controls how long to wait after receiving no new heads before marking the node as out-of-sync.

#### PollFailureThreshold

```
toml
PollFailureThreshold = 5 Default
```

`PollFailureThreshold` indicates how many consecutive polls must fail in order to mark a node as unreachable.

Set to zero to disable poll checking.

#### PollInterval

```
toml
PollInterval = '10s' Default
```

`PollInterval` controls how often to poll the node to check for liveness.

Set to zero to disable poll checking.

#### SelectionMode

```
toml
SelectionMode = 'HighestHead' Default
```

`SelectionMode` controls node selection strategy:

- `HighestHead`: use the node with the highest head number
- `RoundRobin`: rotate through nodes, per-request
- `PriorityLevel`: use the node with the smallest order number
- `TotalDifficulty`: use the node with the greatest total difficulty

#### SyncThreshold

```
toml
SyncThreshold = 5 Default
```

`SyncThreshold` controls how far a node may lag behind the best node before being marked out-of-sync.

Depending on `SelectionMode`, this represents a difference in the number of blocks (`HighestHead`, `RoundRobin`, `PriorityLevel`), or total difficulty (`TotalDifficulty`).

Set to 0 to disable this check.

## LeaseDuration

```
toml
LeaseDuration = '0s' Default
```

LeaseDuration is the minimum duration that the selected "best" node (as defined by SelectionMode) will be used, before switching to a better one if available. It also controls how often the lease check is done.

Setting this to a low value (under 1m) might cause RPC to switch too aggressively.

Recommended value is over 5m

Set to '0s' to disable

## NodeIsSyncingEnabled

```
toml
NodeIsSyncingEnabled = false Default
```

NodeIsSyncingEnabled is a flag that enables syncing health check on each reconnection to an RPC.

Node transitions and remains in Syncing state while RPC signals this state (In case of Ethereum ethsyncing returns anything other than false).

All of the requests to node in state Syncing are rejected.

Set true to enable this check

## FinalizedBlockPollInterval

```
toml
FinalizedBlockPollInterval = '5s' Default
```

FinalizedBlockPollInterval controls how often to poll RPC for new finalized blocks.

The finalized block is only used to report to the poolrpcnodehighestfinalizedblock metric. We plan to use it in RPCs health assessment in the future.

If FinalityTagEnabled = false, poll is not performed and poolrpcnodehighestfinalizedblock is reported based on latest block and finality depth.

Set to 0 to disable.

## EnforceRepeatableRead

```
toml
EnforceRepeatableRead = false Default
```

EnforceRepeatableRead defines if Core should only use RPCs whose most recently finalized block is greater or equal to highest finalized block - FinalizedBlockOffset. In other words, exclude RPCs lagging on latest finalized block.

Set false to disable

## DeathDeclarationDelay

```
toml
```

DeathDeclarationDelay = '10s' Default

DeathDeclarationDelay defines the minimum duration an RPC must be in unhealthy state before producing an error log message. Larger values might be helpful to reduce the noisiness of health checks like `EnforceRepeatableRead = true`, which might be falsely trigger declaration of `FinalizedBlockOutOfSync` due to insignificant network delays in broadcasting of the finalized state among RPCs. RPC will not be picked to handle a request even if this option is set to a nonzero value.

EVM.NodePool.Errors

⚠ ADVANCED: Do not change these settings unless you know what you are doing.

```
toml
[EVM.NodePool.Errors]
NonceTooLow = '(:|^)nonce too low' Example
NonceTooHigh = '(:|^)nonce too high' Example
ReplacementTransactionUnderpriced = '(:|^)replacement transaction underpriced'
Example
LimitReached = '(:|^)limit reached' Example
TransactionAlreadyInMempool = '(:|^)transaction already in mempool' Example
TerminallyUnderpriced = '(:|^)terminally underpriced' Example
InsufficientEth = '(:|^)insufficeint eth' Example
TxFeeExceedsCap = '(:|^)tx fee exceeds cap' Example
L2FeeTooLow = '(:|^)l2 fee too low' Example
L2FeeTooHigh = '(:|^)l2 fee too high' Example
L2Full = '(:|^)l2 full' Example
TransactionAlreadyMined = '(:|^)transaction already mined' Example
Fatal = '(:|^)fatal' Example
ServiceUnavailable = '(:|^)service unavailable' Example
```

Errors enable the node to provide custom regex patterns to match against error messages from RPCs.

NonceTooLow

```
toml
NonceTooLow = '(:|^)nonce too low' Example
```

NonceTooLow is a regex pattern to match against nonce too low errors.

NonceTooHigh

```
toml
NonceTooHigh = '(:|^)nonce too high' Example
```

NonceTooHigh is a regex pattern to match against nonce too high errors.

ReplacementTransactionUnderpriced

```
toml
ReplacementTransactionUnderpriced = '(:|^)replacement transaction underpriced'
Example
```

ReplacementTransactionUnderpriced is a regex pattern to match against replacement transaction underpriced errors.

LimitReached

```
toml
LimitReached = '(:|^)limit reached' Example
```

LimitReached is a regex pattern to match against limit reached errors.

TransactionAlreadyInMempool

```
toml
TransactionAlreadyInMempool = '(:|^)transaction already in mempool' Example
```

TransactionAlreadyInMempool is a regex pattern to match against transaction already in mempool errors.

TerminallyUnderpriced

```
toml
TerminallyUnderpriced = '(:|^)terminally underpriced' Example
```

TerminallyUnderpriced is a regex pattern to match against terminally underpriced errors.

InsufficientEth

```
toml
InsufficientEth = '(:|^)insufficeint eth' Example
```

InsufficientEth is a regex pattern to match against insufficient eth errors.

TxFeeExceedsCap

```
toml
TxFeeExceedsCap = '(:|^)tx fee exceeds cap' Example
```

TxFeeExceedsCap is a regex pattern to match against tx fee exceeds cap errors.

L2FeeTooLow

```
toml
L2FeeTooLow = '(:|^)l2 fee too low' Example
```

L2FeeTooLow is a regex pattern to match against l2 fee too low errors.

L2FeeTooHigh

```
toml
L2FeeTooHigh = '(:|^)l2 fee too high' Example
```

L2FeeTooHigh is a regex pattern to match against l2 fee too high errors.

L2Full

```
toml
L2Full = '(:|^)l2 full' Example
```

L2Full is a regex pattern to match against l2 full errors.

TransactionAlreadyMined

toml

TransactionAlreadyMined = '(:|^)transaction already mined' Example

TransactionAlreadyMined is a regex pattern to match against transaction already mined errors.

Fatal

toml

Fatal = '(:|^)fatal' Example

Fatal is a regex pattern to match against fatal errors.

ServiceUnavailable

toml

ServiceUnavailable = '(:|^)service unavailable' Example

ServiceUnavailable is a regex pattern to match against service unavailable errors.

EVM.OCR

toml

[EVM.OCR]

ContractConfirmations = 4 Default

ContractTransmitterTransmitTimeout = '10s' Default

DatabaseTimeout = '10s' Default

DeltaCOverride = "168h" Default

DeltaCJitterOverride = "1h" Default

ObservationGracePeriod = '1s' Default

ContractConfirmations

toml

ContractConfirmations = 4 Default

ContractConfirmations sets OCR.ContractConfirmations for this EVM chain.

ContractTransmitterTransmitTimeout

toml

ContractTransmitterTransmitTimeout = '10s' Default

ContractTransmitterTransmitTimeout sets OCR.ContractTransmitterTransmitTimeout for this EVM chain.

DatabaseTimeout

toml

DatabaseTimeout = '10s' Default

DatabaseTimeout sets OCR.DatabaseTimeout for this EVM chain.

#### DeltaCOverride

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

toml

DeltaCOverride = "168h" Default

DeltaCOverride (and DeltaCJitterOverride) determine the config override DeltaC. DeltaC is the maximum age of the latest report in the contract. If the maximum age is exceeded, a new report will be created by the report generation protocol.

#### DeltaCJitterOverride

⚠ ADVANCED: Do not change this setting unless you know what you are doing.

toml

DeltaCJitterOverride = "1h" Default

DeltaCJitterOverride is the range for jitter to add to DeltaCOverride.

#### ObservationGracePeriod

toml

ObservationGracePeriod = '1s' Default

ObservationGracePeriod sets OCR.ObservationGracePeriod for this EVM chain.

#### EVM.Nodes

toml

[[EVM.Nodes]]

Name = 'foo' Example

WSURL = 'wss://web.socket/test' Example

HTTPURL = 'https://foo.web' Example

SendOnly = false Default

Order = 100 Default

#### Name

toml

Name = 'foo' Example

Name is a unique (per-chain) identifier for this node.

#### WSURL

toml

WSURL = 'wss://web.socket/test' Example

WSURL is the WS(S) endpoint for this node. Required for primary nodes.

#### HTTPURL

toml

HTTPURL = 'https://foo.web' Example



HTTPURL is the HTTP(S) endpoint for this node. Required for all nodes.

SendOnly

```
toml
SendOnly = false Default
```

SendOnly limits usage to sending transaction broadcasts only. With this enabled, only HTTPURL is required, and WSURL is not used.

Order

```
toml
Order = 100 Default
```

Order of the node in the pool, will takes effect if SelectionMode is PriorityLevel or will be used as a tie-breaker for HighestHead and TotalDifficulty

EVM.OCR2.Automation

```
toml
[EVM.OCR2.Automation]
GasLimit = 5400000 Default
```

GasLimit

```
toml
GasLimit = 5400000 Default
```

GasLimit controls the gas limit for transmit transactions from ocr2automation job.

EVM.Workflow

```
toml
[EVM.Workflow]
FromAddress = '0x2a3e23c6f242F5345320814aC8a1b4E58707D292' Example
ForwarderAddress = '0x2a3e23c6f242F5345320814aC8a1b4E58707D292' Example
```

FromAddress

```
toml
FromAddress = '0x2a3e23c6f242F5345320814aC8a1b4E58707D292' Example
```

FromAddress is Address of the transmitter key to use for workflow writes.

ForwarderAddress

```
toml
ForwarderAddress = '0x2a3e23c6f242F5345320814aC8a1b4E58707D292' Example
```

ForwarderAddress is the keystone forwarder contract address on chain.

Cosmos

```
toml
[[Cosmos]]
ChainID = 'Malaga-420' Example
Enabled = true Default
Bech32Prefix = 'wasm' Default
BlockRate = '6s' Default
BlocksUntilTxTimeout = 30 Default
ConfirmPollPeriod = '1s' Default
FallbackGasPrice = '0.015' Default
GasToken = 'ucosm' Default
GasLimitMultiplier = '1.5' Default
MaxMsgsPerBatch = 100 Default
OCR2CachePollPeriod = '4s' Default
OCR2CacheTTL = '1m' Default
TxMsgTimeout = '10m' Default
```

ChainID

```
toml
ChainID = 'Malaga-420' Example
```

ChainID is the Cosmos chain ID. Mandatory.

Enabled

```
toml
Enabled = true Default
```

Enabled enables this chain.

Bech32Prefix

```
toml
Bech32Prefix = 'wasm' Default
```

Bech32Prefix is the human-readable prefix for addresses on this Cosmos chain.  
See <https://docs.cosmos.network/v0.47/spec/addresses/bech32>.

BlockRate

```
toml
BlockRate = '6s' Default
```

BlockRate is the average time between blocks.

BlocksUntilTxTimeout

```
toml
BlocksUntilTxTimeout = 30 Default
```

BlocksUntilTxTimeout is the number of blocks to wait before giving up on the tx getting confirmed.

ConfirmPollPeriod

```
toml
ConfirmPollPeriod = '1s' Default
```

ConfirmPollPeriod sets how often check for tx confirmation.

FallbackGasPrice

```
toml
FallbackGasPrice = '0.015' Default
```

FallbackGasPrice sets a fallback gas price to use when the estimator is not available.

GasToken

```
toml
GasToken = 'ucosm' Default
```

GasToken is the token denomination which is being used to pay gas fees on this chain.

GasLimitMultiplier

```
toml
GasLimitMultiplier = '1.5' Default
```

GasLimitMultiplier scales the estimated gas limit.

MaxMsgsPerBatch

```
toml
MaxMsgsPerBatch = 100 Default
```

MaxMsgsPerBatch limits the numbers of messages per transaction batch.

OCR2CachePollPeriod

```
toml
OCR2CachePollPeriod = '4s' Default
```

OCR2CachePollPeriod is the rate to poll for the OCR2 state cache.

OCR2CacheTTL

```
toml
OCR2CacheTTL = '1m' Default
```

OCR2CacheTTL is the stale OCR2 cache deadline.

TxMsgTimeout

```
toml
TxMsgTimeout = '10m' Default
```

TxMsgTimeout is the maximum age for resending transaction before they expire.

Cosmos.Nodes

```
toml
[[Cosmos.Nodes]]
Name = 'primary' Example
TendermintURL = 'http://tender.mint' Example
```

Name

```
toml
Name = 'primary' Example
```

Name is a unique (per-chain) identifier for this node.

TendermintURL

```
toml
TendermintURL = 'http://tender.mint' Example
```

TendermintURL is the HTTP(S) tendermint endpoint for this node.

Solana

```
toml
[[Solana]]
ChainID = 'mainnet' Example
Enabled = false Default
BalancePollPeriod = '5s' Default
ConfirmPollPeriod = '500ms' Default
OCR2CachePollPeriod = '1s' Default
OCR2CacheTTL = '1m' Default
TxTimeout = '1m' Default
TxRetryTimeout = '10s' Default
TxConfirmTimeout = '30s' Default
SkipPreflight = true Default
Commitment = 'confirmed' Default
MaxRetries = 0 Default
FeeEstimatorMode = 'fixed' Default
ComputeUnitPriceMax = 1000 Default
ComputeUnitPriceMin = 0 Default
ComputeUnitPriceDefault = 0 Default
FeeBumpPeriod = '3s' Default
BlockHistoryPollPeriod = '5s' Default
```

ChainID

```
toml
ChainID = 'mainnet' Example
```

ChainID is the Solana chain ID. Must be one of: mainnet, testnet, devnet, localnet. Mandatory.

Enabled

```
toml
Enabled = false Default
```

Enabled enables this chain.

BalancePollPeriod

```
toml
BalancePollPeriod = '5s' Default
```

BalancePollPeriod is the rate to poll for SOL balance and update Prometheus metrics.

ConfirmPollPeriod

```
toml
ConfirmPollPeriod = '500ms' Default
```

ConfirmPollPeriod is the rate to poll for signature confirmation.

OCR2CachePollPeriod

```
toml
OCR2CachePollPeriod = '1s' Default
```

OCR2CachePollPeriod is the rate to poll for the OCR2 state cache.

OCR2CacheTTL

```
toml
OCR2CacheTTL = '1m' Default
```

OCR2CacheTTL is the stale OCR2 cache deadline.

TxTimeout

```
toml
TxTimeout = '1m' Default
```

TxTimeout is the timeout for sending txes to an RPC endpoint.

TxRetryTimeout

```
toml
TxRetryTimeout = '10s' Default
```

TxRetryTimeout is the duration for tx manager to attempt rebroadcasting to RPC, before giving up.

TxConfirmTimeout

```
toml
TxConfirmTimeout = '30s' Default
```

TxConfirmTimeout is the duration to wait when confirming a tx signature, before discarding as unconfirmed.

SkipPreflight

```
toml
SkipPreflight = true Default
```

SkipPreflight enables or disables preflight checks when sending txs.

Commitment

toml

Commitment = 'confirmed' Default

Commitment is the confirmation level for solana state and transactions.  
(documentation)

MaxRetries

toml

MaxRetries = 0 Default

MaxRetries is the maximum number of times the RPC node will automatically rebroadcast a tx.  
The default is 0 for custom txm rebroadcasting method, set to -1 to use the RPC node's default retry strategy.

FeeEstimatorMode

toml

FeeEstimatorMode = 'fixed' Default

FeeEstimatorMode is the method used to determine the base fee

ComputeUnitPriceMax

toml

ComputeUnitPriceMax = 1000 Default

ComputeUnitPriceMax is the maximum price per compute unit that a transaction can be bumped to

ComputeUnitPriceMin

toml

ComputeUnitPriceMin = 0 Default

ComputeUnitPriceMin is the minimum price per compute unit that transaction can have

ComputeUnitPriceDefault

toml

ComputeUnitPriceDefault = 0 Default

ComputeUnitPriceDefault is the default price per compute unit price, and the starting base fee when FeeEstimatorMode = 'fixed'

FeeBumpPeriod

toml

FeeBumpPeriod = '3s' Default

FeeBumpPeriod is the amount of time before a tx is retried with a fee bump

## BlockHistoryPollPeriod

```
toml
BlockHistoryPollPeriod = '5s' Default
```

BlockHistoryPollPeriod is the rate to poll for blocks in the block history fee estimator

## Solana.Nodes

```
toml
[[Solana.Nodes]]
Name = 'primary' Example
URL = 'http://solana.web' Example
```

### Name

```
toml
Name = 'primary' Example
```

Name is a unique (per-chain) identifier for this node.

### URL

```
toml
URL = 'http://solana.web' Example
```

URL is the HTTP(S) endpoint for this node.

## Starknet

```
toml
[[Starknet]]
ChainID = 'foobar' Example
FeederURL = 'http://feeder.url' Example
Enabled = true Default
OCR2CachePollPeriod = '5s' Default
OCR2CacheTTL = '1m' Default
RequestTimeout = '10s' Default
TxTimeout = '10s' Default
ConfirmationPoll = '5s' Default
```

### ChainID

```
toml
ChainID = 'foobar' Example
```

ChainID is the Starknet chain ID.

### FeederURL

```
toml
FeederURL = 'http://feeder.url' Example
```

FeederURL is required to get tx metadata (that the RPC can't)

Enabled

```
toml
Enabled = true Default
```

Enabled enables this chain.

OCR2CachePollPeriod

```
toml
OCR2CachePollPeriod = '5s' Default
```

OCR2CachePollPeriod is the rate to poll for the OCR2 state cache.

OCR2CacheTTL

```
toml
OCR2CacheTTL = '1m' Default
```

OCR2CacheTTL is the stale OCR2 cache deadline.

RequestTimeout

```
toml
RequestTimeout = '10s' Default
```

RequestTimeout is the RPC client timeout.

TxTimeout

```
toml
TxTimeout = '10s' Default
```

TxTimeout is the timeout for sending txes to an RPC endpoint.

ConfirmationPoll

```
toml
ConfirmationPoll = '5s' Default
```

ConfirmationPoll is how often to confirmers checks for tx inclusion on chain.

Starknet.Nodes

```
toml
[[Starknet.Nodes]]
Name = 'primary' Example
URL = 'http://stark.node' Example
APIKey = 'key' Example
```

Name

```
toml
Name = 'primary' Example
```

Name is a unique (per-chain) identifier for this node.



URL

```
toml
URL = 'http://stark.node' Example
```

URL is the base HTTP(S) endpoint for this node.

APIKey

```
toml
APIKey = 'key' Example
```

APIKey Header is optional and only required for Nethermind RPCs

```
roles-and-access.mdx:

section: nodeOperator
date: Last Modified
title: "Role-Based Access Control (RBAC)"

```

Chainlink Nodes allow the root admin CLI user and any additional admin users to create and assign tiers of role-based access to new users. These new API users can able to log in to the Operator UI independently.

Each user has a specific role assigned to their account. There are four roles: admin, edit, run, and view.

If there are multiple users who need specific access to manage the Chainlink Node instance, permissions and level of access can be set here.

User management is configured through the use of the admin chainlink admin users command. Run chainlink admin login before you set user roles for other accounts. For example, a view-only user can be created with the following command:

```
shell
chainlink admin users create --email=operator-ui-view-only@test.com --role=view
```

To modify permissions or delete existing users, run the admin users chrole or admin users delete commands. Use the -h flag to get a full list of options for these commands:

```
shell
chainlink admin users chrole -h
```

```
shell
chainlink admin users delete -h
```

Specific actions are enabled to check role-based access before they execute. The following table lists the actions that have role-based access and the role that is required to run that action:

Action	View	Run	Edit	Admin
Update password	X	X	X	X
Create self API token	X	X	X	X

Delete self API token	X	X	X	X
List external initiators	X	X	X	X
Create external initiator			X	X
Delete external initiator			X	X
List bridges	X	X	X	X
View bridge	X	X	X	X
Create bridge			X	X
Edit bridge			X	X
Delete bridge			X	X
View config	X	X	X	X
Update config				X
Dump env/config				X
View transaction attempts	X	X	X	X
View transaction attempts EVM	X	X	X	X
View transactions	X	X	X	X
Replay a specific block number		X	X	X
List keys (CSA,ETH,OCR(2),P2P,Solana,Terra)	X	X	X	X
Create keys (CSA,ETH,OCR(2),P2P,Solana,Terra)			X	X
Delete keys (CSA,ETH,OCR(2),P2P,Solana,Terra)				X
Import keys (CSA,ETH,OCR(2),P2P,Solana,Terra)				X
Export keys (CSA,ETH,OCR(2),P2P,Solana,Terra)				X
List jobs	X	X	X	X
View job	X	X	X	X
Create job			X	X
Delete job			X	X
List pipeline runs	X	X	X	X
View job runs	X	X	X	X
Delete job spec errors			X	X
View features	X	X	X	X
View log	X	X	X	X
Update log				X
List chains	X	X	X	X
View chain	X	X	X	X
Create chain			X	X
Update chain			X	X
Delete chain			X	X
View nodes	X	X	X	X
Create node			X	X
Update node			X	X
Delete node			X	X
View forwarders	X	X	X	X
Create forwarder			X	X
Delete forwarder			X	X
Create job run		X	X	X
Create Transfer EVM				X
Create Transfer Terra				X
Create Transfer Solana				X
Create user				X
Delete user				X
Edit user				X
List users				X

The run command allows for minimal interaction and only enables the ability to replay a specific block number and kick off a job run.

# running-a-chainlink-node.mdx:

```

section: nodeOperator
date: Last Modified
title: "Running a Chainlink Node"
whatsnext:
{

```

```

 "Fulfilling Requests": "/chainlink-nodes/v1/fulfilling-requests",
 "Role-Based Access Control": "/chainlink-nodes/v1/roles-and-access",
 "Requirements": "/chainlink-nodes/resources/requirements",
 "Optimizing EVM Performance": "/chainlink-nodes/resources/evm-performance-configuration",
 "Performing System Maintenance": "/chainlink-nodes/resources/performing-system-maintenance",
 "Miscellaneous": "/chainlink-nodes/resources/miscellaneous",
 "Security and Operation Best Practices": "/chainlink-nodes/resources/best-security-practices",
 }
 metadata:
 title: "Running a Chainlink Node locally"
 description: "Run your own Chainlink node using this guide which explains the requirements and basics for getting started."

```

```

import { Aside } from "@components"
import { Tabs } from "@components/Tabs"

```

This guide will teach you how to run a Chainlink node locally using Docker. The Chainlink node will be configured to connect to the Ethereum Sepolia.

```

<Aside type="note" title="Running from source">
 To run a Chainlink node from source, use the following
 instructions. However, it's recommended to run the Chainlink
 node with Docker. This is because we continuously build and deploy the code
 from our repository on Github, which means
 you don't need a complete development environment to run a node.
</Aside>

```

```

<Aside type="note" title="Supported networks">
 Chainlink is a blockchain agnostic technology. The LINK Token Contracts page
 details networks which support the LINK token. You can set up your node to
 provide data to any of these blockchains.

```

Ganache is a mock testnet. Although you can run nodes on Ganache, it is not officially supported. Most node operators should use one of the supported testnets for development and testing.

```

</Aside>

```

## Requirements

- As explained in the requirements page, make sure there are enough resources to run a Chainlink node and a PostgreSQL database.
- Install Docker Desktop. You will run the Chainlink node and PostgreSQL in Docker containers.
- Chainlink nodes must be able to connect to an Ethereum client with an active websocket connection. See Running an Ethereum Client for details. In this tutorial, you can use an external service as your client.

## Using Docker

### Run PostgreSQL

1. Run PostgreSQL in a Docker container. You can replace mysecretpassword with your own password.

```

shell
docker run --name cl-postgres -e POSTGRESPASSWORD=mysecretpassword -p
5432:5432 -d postgres

```

1. Confirm that the container is running. Note the 5432 port is published 0.0.0.0:5432->5432/tcp and therefore accessible outside of Docker.

```
shell
docker ps -a -f name=cl-postgres
```

If the container is running successfully, the output shows a healthy status:

```
shell
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
dc08cfad2a16 postgres "docker-entrypoint.sâ |" 3 minutes ago Up 3
minutes 0.0.0.0:5432->5432/tcp cl-postgres
```

Run Chainlink node

Configure your node

1. Create a local directory to hold the Chainlink data:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 mkdir /.chainlink-sepolia
 </Fragment>
</Tabs>
```

1. Run the following as a command to create a config.toml file and populate with variables specific to the network you're running on. For a full list of available configuration variables, see the Node Config page.

Be sure to update the value for CHANGEME to the value given by your external Ethereum provider.

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 echo "[Log]
 Level = 'warn'

 [WebServer]
 AllowOrigins = '\
 SecureCookies = false

 [WebServer.TLS]
 HTTPSPort = 0

 [[EVM]]
 ChainID = '11155111'

 [[EVM.Nodes]]
 Name = 'Sepolia'
 WSURL = 'wss://CHANGEME'
 HTTPURL = 'https://CHANGEME'
 " > /.chainlink-sepolia/config.toml
 </Fragment>
```

</Tabs>

1. Create a secrets.toml file with a keystore password and the URL to your database. Update the value for mysecretpassword to the chosen password in Run PostgreSQL. Specify a complex keystore password. This will be your wallet password that you can use to unlock the keystore file generated for you.

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 echo "[Password]
 Keystore = 'mysecretkeystorepassword'
 [Database]
 URL =
'postgresql://postgres:mysecretpassword@host.docker.internal:5432/postgres?
sslmode=disable'
 " > /.chainlink-sepolia/secrets.toml

 </Fragment>
</Tabs>

<Aside type="tip" title="Important">
 Because you are testing locally, add ?sslmode=disable to the end
of your DATABASEURL. However you should never
 do this on a production node.
</Aside>
```

1. Optionally, you can create an .api file with the credentials for the node's API and Operator Interface. The node stores the credentials from the .api file in the database only the first time you run the container using the database. The .api file cannot override credentials for an existing user in the database.

Create the file in the same directory as your TOML config files and list your API credentials. Change the values for API email and password. The user must be an email address with an @ character and the password must be 16-50 characters in length.

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 echo "CHANGETHISEXAMPLEEMAIL
 CHANGETHISEXAMPLEPASSWORD
 " > /.chainlink-sepolia/.api

 </Fragment>
</Tabs>
```

1. Start the Chainlink Node by running the Docker image.

Change the version number in smartcontract/chainlink:2.15.0 with the version of the Docker image that you need to run. For most new nodes, use version 2.0.0 or later. Tag versions are available in the Chainlink Docker hub. The latest version does not work.

Chainlink Nodes running 2.0.0 and later require the -config and -secrets flags after the node part of the command.

If you created an .api file with your API and Operator UI login credentials, add -a /chainlink/.api to the end of the docker run command. Otherwise, the node will ask you for these credentials when you start it for the first time. These

credentials are stored in the database only when you run a container for the first time against that database. If you need to remove the .api file, delete the container, and start it again without -a /chainlink/.api.

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.1">Sepolia</Fragment>
 <Fragment slot="panel.1">
 shell Sepolia
 cd /.chainlink-sepolia && docker run --platform linux/x8664/v8
--name chainlink -v /.chainlink-sepolia:/chainlink -it -p 6688:6688 --add-
host=host.docker.internal:host-gateway smartcontract/chainlink:2.15.0 node -
config /chainlink/config.toml -secrets /chainlink/secrets.toml start

 </Fragment>
</Tabs>
```

1. Detach from the container by pressing the Ctrl+P command and then the Ctrl-Q command. On MacOS, use `â€-P` and `â€-Q`.

1. Confirm that the container is running. Note that the 6688 port is published 0.0.0.0:6688->6688/tcp and is accessible outside of Docker.

```
shell
docker ps -a -f name=chainlink
```

If the container is running, the output shows a healthy status:

```
shell
CONTAINER ID IMAGE COMMAND
CREATED STATUS PORTS
NAMES
867e792d6f78 smartcontract/chainlink:2.15.0 "chainlink node -conâ€|" 2
minutes ago Up 2 minutes (healthy) 0.0.0.0:6688->6688/tcp, :::6688->6688/tcp
chainlink
```

1. You can now connect to your Chainlink node's UI interface by navigating to <http://localhost:6688>. Use the API credentials you set up earlier to log in.

If you are using a VPS, you can create an SSH tunnel to your node for 6688:localhost:6688 to enable connectivity to the GUI. Typically this is done with `ssh -i $KEY $USER@$REMOTE-IP -L 6688:localhost:6688 -N`. An SSH tunnel is recommended over opening public-facing ports specific to the Chainlink node. See the Security and Operation Best Practices page for more details about securing your node.

## Configure users and roles

You can create several users with different role-based access tiers. This allows you to grant access to several users without granting admin privileges to every user. Role-based access can be configured only by using the CLI.

1. Open an interactive bash shell on the container that is running your node:

```
shell
docker exec -it chainlink /bin/bash
```

1. Log into the Chainlink CLI. The CLI prompts you for the admin credentials that you configured for your node.

```
shell
chainlink admin login
```

1. Add a user with view-only permissions on the node. The CLI prompts you for the new user's credentials.

```
shell
chainlink admin users create --email=operator-ui-view-only@test.com --
role=view
```

This user can now log into the UI and query the API, but cannot change any settings or jobs.

1. Confirm the current list of users:

```
shell
chainlink admin users list
```

1. Log out of the CLI. This prevents users with access to the shell from executing admin commands.

```
shell
chainlink admin logout
```

1. Exit from the container.

```
shell
exit
```

To learn how to modify user roles and see the full list of available roles, read the Role-Based Access Control page.

# secrets-config.mdx:

```

section: nodeOperator
date: Last Modified
title: "Secrets Config (TOML)"

```

[//]: "Documentation generated from docs/secrets.toml - DO NOT EDIT."

This document describes the TOML format for secrets.

Each secret has an alternative corresponding environment variable.

See also: Node Config

Example

```
toml
[Database]
URL = 'postgresql://user:pass@localhost:5432/dbname?sslmode=disable' Required

[Password]
Keystore = 'keystorepass' Required
```

## Database

```
toml
[Database]
URL = "postgresql://user:pass@localhost:5432/dbname?sslmode=disable" Example
BackupURL = "postgresql://user:pass@read-replica.example.com:5432/dbname?
sslmode=disable" Example
AllowSimplePasswords = false Default
```

## URL

```
toml
URL = "postgresql://user:pass@localhost:5432/dbname?sslmode=disable" Example
```

URL is the PostgreSQL URI to connect to your database. Chainlink nodes require Postgres versions  $\geq 11$ . See [Running a Chainlink Node](#) for an example.

Environment variable: CLDATABASEURL

## BackupURL

```
toml
BackupURL = "postgresql://user:pass@read-replica.example.com:5432/dbname?
sslmode=disable" Example
```

BackupURL is where the automatic database backup will pull from, rather than the main CLDATABASEURL. It is recommended to set this value to a read replica if you have one to avoid excessive load on the main database.

Environment variable: CLDATABASEBACKUPURL

## AllowSimplePasswords

```
toml
AllowSimplePasswords = false Default
```

AllowSimplePasswords skips the password complexity check normally enforced on URL & BackupURL.

Environment variable: CLDATABASEALLOWSIMPLEPASSWORDS

## WebServer.LDAP

```
toml
[WebServer.LDAP]
ServerAddress = 'ldaps://127.0.0.1' Example
ReadOnlyUserLogin = 'viewer@example.com' Example
ReadOnlyUserPass = 'password' Example
```

## Optional LDAP config

### ServerAddress

```
toml
ServerAddress = 'ldaps://127.0.0.1' Example
```



ServerAddress is the full ldaps:// address of the ldap server to authenticate with and query

ReadOnlyUserLogin

```
toml
ReadOnlyUserLogin = 'viewer@example.com' Example
```

ReadOnlyUserLogin is the username of the read only root user used to authenticate the requested LDAP queries

ReadOnlyUserPass

```
toml
ReadOnlyUserPass = 'password' Example
```

ReadOnlyUserPass is the password for the above account

Password

```
toml
[Password]
Keystore = "keystorepass" Example
VRF = "VRFpass" Example
```

Keystore

```
toml
Keystore = "keystorepass" Example
```

Keystore is the password for the node's account.

Environment variable: CLPASSWORDKEystore

VRF

```
toml
VRF = "VRFpass" Example
```

VRF is the password for the vrf keys.

Environment variable: CLPASSWORDVRF

Pyroscope

```
toml
[Pyroscope]
AuthToken = "pyroscope-token" Example
```

AuthToken

```
toml
AuthToken = "pyroscope-token" Example
```

AuthToken is the API key for the Pyroscope server.

Environment variable: CLPYROSCOPEAUTHTOKEN

## Prometheus

```
toml
[Prometheus]
AuthToken = "prometheus-token" Example
```

### AuthToken

```
toml
AuthToken = "prometheus-token" Example
```

AuthToken is the authorization key for the Prometheus metrics endpoint.

Environment variable: CLPROMETHEUSAUTHTOKEN

### Mercury.Credentials.Name

```
toml
[Mercury.Credentials.Name]
Username = "A-Mercury-Username" Example
Password = "A-Mercury-Password" Example
URL = "https://example.com" Example
LegacyURL = "https://example.v1.com" Example
```

### Username

```
toml
Username = "A-Mercury-Username" Example
```

Username is used for basic auth of the Mercury endpoint

### Password

```
toml
Password = "A-Mercury-Password" Example
```

Password is used for basic auth of the Mercury endpoint

### URL

```
toml
URL = "https://example.com" Example
```

URL is the Mercury endpoint base URL used to access Mercury price feed

### LegacyURL

```
toml
LegacyURL = "https://example.v1.com" Example
```

LegacyURL is the Mercury legacy endpoint base URL used to access Mercury v0.2 price feed

### Threshold

```
toml
```

```
[Threshold]
ThresholdKeyShare = "A-Threshold-Decryption-Key-Share" Example
```

ThresholdKeyShare

```
toml
ThresholdKeyShare = "A-Threshold-Decryption-Key-Share" Example
```

ThresholdKeyShare used by the threshold decryption OCR plugin

```
using-forwarder.mdx:
```

```

section: nodeOperator
date: Last Modified
title: "Forwarder tutorial"
metadata:
 title: "Chainlink Node Operators: Forwarder tutorial"
 description: "Use a forwarder contract for more security and flexibility."

```

```
import { Aside, ClickToZoom, CodeSample } from "@components"
```

```
<Aside type="note" title="Prerequisites">
This guide assumes you have a running Chainlink node. See the Running a
Chainlink Node locally guide to learn how to run a Chainlink node. Note: For
this example to work, you must use Chainlink node version 1.12.0 or above.
```

Also, you must be familiar with these concepts:

- Forwarder contracts.
- Operator contracts.
- OperatorFactory contracts.

```
</Aside>
```

```
<ClickToZoom src="/images/chainlink-nodes/node-operators/forwarder/forwarder-
directrequest-example.webp" />
```

In this tutorial, you will configure your Chainlink node with a simple transaction-sending strategy on the Sepolia testnet:

- Your node has two externally owned accounts (EOA).
- Your node has two direct request jobs. One job returns uint256, and the other returns string.
- Each job uses a different EOA.
- You use a forwarder contract to fulfill requests with two EOAs that look like a single address.

```
<Aside type="note">
 Here you are using a forwarder contract and two EOAs for two direct
 request jobs. You can use the same strategy for different job
 types, VRF and OCR. Supporting different job types on the same Chainlink node
 reduces your infrastructure and
 maintenance costs.
</Aside>
```

Check your Chainlink node is running

On your terminal, check that your Chainlink node is running:

```
shell
docker ps -a -f name=chainlink
```

CONTAINER ID	IMAGE	COMMAND NAMES	CREATED
feff39f340d6	smartcontract/chainlink:1.12.0	"chainlink local n"	4 minutes ago
Up 4 minutes (healthy)	0.0.0.0:6688->6688/tcp	chainlink	

Your Chainlink Operator Interface is accessible on <http://localhost:6688>. If using a VPS, you can create a SSH tunnel to your node for 6688:localhost:6688 to enable connectivity to the GUI. Typically this is done with `ssh -i $KEY $USER@$REMOTE-IP -L 6688:localhost:6688 -N`. A SSH tunnel is recommended over opening up ports specific to the Chainlink node to be public facing. See the Security and Operation Best Practices page for more details on how to secure your node.

If you don't have a running Chainlink node, follow the Running a Chainlink Node Locally guide.

Set up multiple EOAs

1. Access the shell of your Chainlink node container:

```
shell
docker exec -it chainlink /bin/bash
```

```
shell
chainlink@1d095e4ceb09:$
```

1. You can now log in by running:

```
shell
chainlink admin login
```

You will be prompted to enter your API email and password. If successful, the prompt will appear again.

1. Check the number of available EOA:

```
shell
chainlink keys eth list
```

You should see one EOA:

```
shell
ðŸ”” · ETH keys
```

```


Address: 0x71a1Eb6534054E75F0D6fD0A3B0A336228DD5cFc
EVM Chain ID: 11155111
Next Nonce: 0
ETH: 0.00000000000000000000
LINK: 0
Disabled: false
Created: 2023-03-02 09:28:26.872791 +0000 UTC
Updated: 2023-03-02 09:28:26.872791 +0000 UTC
Max Gas Price Wei:
115792089237316195423570985008687907853269984665640564039457584007913129639935ch
```

ainlink@22480bec8986

1. Create a new EOA for your Chainlink node:

```
shell
chainlink keys eth create
```

```
shell
ETH key created.
```

```
ðŸ”” ‘ New key
```

```


Address: 0x259c49E65644a020C2A642260a4ffb0CD862cb24
EVM Chain ID: 11155111
Next Nonce: 0
ETH: 0.00000000000000000000
LINK: 0
Disabled: false
Created: 2023-03-02 11:36:48.717074 +0000 UTC
Updated: 2023-03-02 11:36:48.717074 +0000 UTC
Max Gas Price Wei:
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

1. At this point, there are two EOAs:

```
shell
chainlink keys eth list
```

```
shell
ðŸ”” ‘ ETH keys
```

```


Address: 0x259c49E65644a020C2A642260a4ffb0CD862cb24
EVM Chain ID: 11155111
Next Nonce: 0
ETH: 0.00000000000000000000
LINK: 0
Disabled: false
Created: 2023-03-02 11:36:48.717074 +0000 UTC
Updated: 2023-03-02 11:36:48.717074 +0000 UTC
Max Gas Price Wei:
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

```


Address: 0x71a1Eb6534054E75F0D6fD0A3B0A336228DD5cFc
EVM Chain ID: 11155111
Next Nonce: 0
ETH: 0.00000000000000000000
LINK: 0
Disabled: false
Created: 2023-03-02 09:28:26.872791 +0000 UTC
Updated: 2023-03-02 09:28:26.872791 +0000 UTC
Max Gas Price Wei:
115792089237316195423570985008687907853269984665640564039457584007913129639935ch
ainlink@22480bec8986
```

1. Fund the two addresses with 0.5 Sepolia ETH each. You can obtain testnet ETH from the faucets listed on the Link Token Contracts page.
1. Note the two addresses, as you will need them later.

## Deploy operator and forwarder

Use the operator factory to deploy both the forwarder and the operator contracts. You can find the factory address for each network on the addresses page.

1. Open contract 0x447Fd5eC2D383091C22B8549cb231a3bAD6d3fAf to display the factory in the Sepolia block explorer.
1. Click the Contract tab. Then, click Write Contract to display the write transactions on the factory.

```
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/factorytransactions.jpg" /
>
```

1. Click the Connect to Web3 button to connect your wallet.

```
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/factorytransactionsconnect
ed.jpg" />
```

1. Click the deployNewOperatorAndForwarder function to expand it and then click the Write button to run the function. Metamask prompts you to confirm the transaction.

1. Click View your transaction. Etherscan will open a new tab. Wait for the transaction to be successful.

```
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/factoryviewtransaction.jpg
" />
```

```
{" "}
```

```
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/deployfromfactorysuccess.j
pg" />
```

1. On the Transaction Details page, click Logs to display the list of transaction events. Notice the OperatorCreated and AuthorizedForwarderCreated events.

```
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/factorycreatetransactions.
jpg" />
```

1. Right-click on each contract address and open it in a new tab.
1. At this point, you should have one tab displaying the operator contract and one tab displaying the forwarder contract.

```
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/operator.jpg" />
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/forwarder.jpg" />
```

1. Record the operator and forwarder addresses. You will need them later.

## Access control setup

As explained in the forwarder page:

- The owner of a forwarder contract is an operator contract. The owner of the operator contract is a more secure address, such as a hardware wallet or a multisig wallet. Therefore, node operators can manage a set of forwarder contracts through an operator contract using a secure account such as hardware or a multisig wallet.
- Forwarder contracts distinguish between owners and authorized senders. Authorized senders are hot wallets (Chainlink nodes' EOAs).

For this example to run, you will have to:

- Allow the forwarder contract to call the operator's `fulfillOracleRequest2` function by calling the `setAuthorizedSenders` function on the operator contract. Specify the forwarder address as a parameter.
- Allow the two Chainlink node EOAs to call the forwarder's `forward` function. Because the operator contract owns the forwarder contract, call `acceptAuthorizedReceivers` on the operator contract. Specify the forwarder contract address and the two Chainlink node EOAs as parameters. This call makes the operator contract accept ownership of the forwarder contract and authorizes the Chainlink node EOAs to call the forwarder contract by calling `setAuthorizedSenders`.

Whitelist the forwarder

In the blockchain explorer, view the operator contract and call the `setAuthorizedSenders` method with the address of your forwarder contract. The parameter is an array. For example, `["0xA3f07D6773514480b918C2742b027b3acD9E44fA"]`. Metamask prompts you to confirm the transaction.

```
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/operatorsetauthorizedsenders.jpg" />
```

Whitelist the Chainlink node EOAs

In the blockchain explorer, view the operator contract and call the `acceptAuthorizedReceivers` method with the following parameters:

- `targets`: Specify an array of forwarder addresses. For example, `["0xA3f07D6773514480b918C2742b027b3acD9E44fA"]`
- `sender`: Specify an array with the two Chainlink node EOAs. For example, `["0x259c49E65644a020C2A642260a4ffb0CD862cb24", "0x71a1Eb6534054E75F0D6fD0A3B0A336228DD5cFc"]`.

MetaMask prompts you to confirm the transaction.

```
<ClickToZoom
src="/images/chainlink-nodes/node-operators/forwarder/acceptAuthorizedReceivers.jpg" />
```

Activate the forwarder

1. In the shell of your Chainlink node, enable the forwarder using the Chainlink CLI. Replace `forwarderAddress` with the forwarder address that was created by the factory. Replace `chainId` with the EVM chain ID. (11155111 for Sepolia):

```
shell
chainlink forwarders track --address forwarderAddress --evmChainID chainId
```

In this example, the command is:

```
shell
chainlink forwarders track --address
0xA3f07D6773514480b918C2742b027b3acD9E44fA --evmChainID 11155111
```

```
shell
Forwarder created

ID: 1
Address: 0xA3f07D6773514480b918C2742b027b3acD9E44fA
Chain ID: 11155111
Created At: 2023-03-02T11:41:43Zchainlink@22480bec8986
```

1. Exit the shell of your Chainlink node:

```
shell
exit
```

1. Stop your Chainlink node:

```
shell
docker stop chainlink && docker rm chainlink
```

1. Update your environment file. If you followed Running a Chainlink Node locally guide then add ETHUSEFORWARDERS=true and FEATURELOGPOLLER=true to your environment file:

```
shell
echo "ETHUSEFORWARDERS=true
FEATURELOGPOLLER=true" >> /.chainlink-sepolia/.env
```

- ETHUSEFORWARDERS enables sending transactions through forwarder contracts.
- FEATURELOGPOLLER enables polling forwarder contracts logs to detect any changes to the authorized senders.

1. Start the Chainlink node:

```
shell
cd /.chainlink-sepolia && docker run --platform linux/x8664/v8 --name
chainlink -v /.chainlink-sepolia:/chainlink -it --env-file=.env -p 6688:6688
smartcontract/chainlink:1.12.0 local n
```

1. You will be prompted to enter the key store password:

```
shell
2022-12-22T16:18:04.706Z [WARN] P2PLISTENPORT was not set, listening on
random port 59763. A new random port will be generated on every boot, for
stability it is recommended to set P2PLISTENPORT to a fixed value in your
environment config/p2pv1config.go:84 logger=1.10.0@aeb8c80.GeneralConfig
p2pPort=59763
2022-12-22T16:18:04.708Z [DEBUG] Offchain reporting disabled
chainlink/application.go:373 logger=1.10.0@aeb8c80
2022-12-22T16:18:04.709Z [DEBUG] Offchain reporting v2 disabled
chainlink/application.go:422 logger=1.10.0@aeb8c80
Enter key store password:
```

1. Enter the key store password and wait for your Chainlink node to start.



## Create directRequest jobs

This section is similar to the Fulfilling Requests guide.

1. Open the Chainlink Operator UI.

1. On the Jobs tab, click New Job.

1. Create the uint256 job. Replace:

- `YOUROPERATORCONTRACTADDRESS` with the address of your deployed operator contract address.

- `EOAADDRESS` with the first Chainlink node EOA.

```
<CodeSample src="samples/ChainlinkNodes/forwarder/get-uint256.toml" />
```

1. Create the string job. Replace:

- `YOUROPERATORCONTRACTADDRESS` with the address of your deployed operator contract address.

- `EOAADDRESS` with the second Chainlink node EOA.

```
<CodeSample src="samples/ChainlinkNodes/forwarder/get-string.toml" />
```

1. After you create the jobs, record the two job IDs.

<Aside type="note">

Note that both jobs have the attribute `forwardingAllowed = true`. This attribute will make the jobs attempt to run through the forwarder before falling back to default mode. For example, if the Chainlink node tracks no forwarder, the job falls back to the default mode.

</Aside>

Test the transaction-sending strategy

## Create API requests

1. Open `APIConsumerForwarder.sol` in the Remix IDE.

1. Note that `setChainlinkToken(0x779877A7B0D9E8603169DdbD7836e478b4624789)` is configured for Sepolia.

1. On the Compiler tab, click the Compile button for `APIConsumerForwarder.sol`.

1. On the Deploy and Run tab, configure the following settings:

- Select Injected Provider as your environment. Make sure your metamask is connected to Sepolia.

- Select `APIConsumerForwarder` from the Contract menu.

1. Click Deploy. MetaMask prompts you to confirm the transaction.

1. Fund the contract by sending LINK to the contract's address. See the Fund Your Contracts page for instructions. The address for the `ATestnetConsumer` contract is on the list of your deployed contracts in Remix. You can fund your contract with 1 LINK.

1. After you fund the contract, create a request. Input your operator contract address and the job ID for the `Get > Uint256` job into the `requestEthereumPrice` request method without dashes. The job ID is the `externalJobID` parameter, which you can find on your job's definition page in the Node Operators UI.

1. Click the transact button for the `requestEthereumPrice` function and approve the transaction in Metamask. The `requestEthereumPrice` function asks the node to retrieve uint256 data specifically from

<https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=USD>.

1. After the transaction processes, open the Runs page in the Node Operators UI. You can see the details for the completed the job run.

1. In Remix, click the currentPrice variable to see the current price updated on your consumer contract.

1. Input your operator contract address and the job ID for the Get > String job into the requestFirstId request method without dashes. The job ID is the externalJobID parameter, which you can find on your job's definition page in the Node Operators UI.

1. Click the transact button for the requestFirstId function and approve the transaction in Metamask. The requestFirstId function asks the node to retrieve the first id from <https://api.coingecko.com/api/v3/coins/markets?vscurrency=usd&perpage=10>.

1. After the transaction processes, you can see the details for the complete the job run the Runs page in the Node Operators UI.

1. In Remix, click the id variable to see the current price updated on your consumer contract.

Check the forwarder

Confirm the following information:

- The Chainlink node submitted the callbacks to the forwarder contract.
- Each callback used a different account.
- The forwarder contract forwarded the callbacks to the operator contract.

Open your forwarder contract in the block explorer. Two forward transactions exist.

<ClickToZoom src="/images/chainlink-nodes/node-operators/forwarder/forward-transactions.jpg" />

Click on both transaction hashes and note that the From addresses differ. In this example, 0x259c49E65644a020C2A642260a4ffb0CD862cb24 is the EOA used in the uint256 job, and 0x71a1Eb6534054E75F0D6fD0A3B0A336228DD5cFc is the EOA used in the string job:

<ClickToZoom src="/images/chainlink-nodes/node-operators/forwarder/forward1.jpg" />

<ClickToZoom src="/images/chainlink-nodes/node-operators/forwarder/forward2.jpg" />

Then click on the Logs tab to display the list of events. Notice the OracleResponse events. The operator contract emitted them when the Forward contract called it

<ClickToZoom src="/images/chainlink-nodes/node-operators/forwarder/forward1-logs.jpg" />

<ClickToZoom src="/images/chainlink-nodes/node-operators/forwarder/forward2-logs.jpg" />

# api-reference.mdx:

---

section: dataFeeds

```

date: Last Modified
title: "Data Feeds API Reference"
metadata:
 description: "API reference for using Chainlink Data Feeds in smart
contracts."

```

```
import { Aside, Icon } from "@components"
```

When you use data feeds, retrieve the feeds through the `AggregatorV3Interface` and the proxy address. Optionally, you can call variables and functions in the `AccessControlledOffchainAggregator` contract to get information about the aggregator behind the proxy.

### AggregatorV3Interface

Import this interface to your contract and use it to run functions in the proxy contract. Create the interface object by pointing to the proxy address. For example, on Sepolia you could create the interface object in the constructor of your contract using the following example:

```

{/ prettier-ignore /}
solidity
/
 Network: Sepolia
 Data Feed: ETH/USD
 Address: 0x694AA1769357215DE4FAC081bf1f309aDC325306
/
constructor() {
 priceFeed = AggregatorV3Interface(0x694AA1769357215DE4FAC081bf1f309aDC325306);
}
```

To see examples for how to use this interface, read the [Using Data Feeds](#) guide.

You can see the code for the `AggregatorV3Interface` contract on [GitHub](#).

### Functions in AggregatorV3Interface

Name	Description
-----	
decimals	The number of decimals in the response.
description	The description of the aggregator that the proxy points to.
getRoundData	Get data from a specific round.
latestRoundData	Get data from the latest round.
version	The version representing the type of aggregator the proxy points to.

#### decimals

Get the number of decimals present in the response value.

```

{/ prettier-ignore /}
solidity
function decimals() external view returns (uint8);
```

- RETURN: The number of decimals.

#### description

Get the description of the underlying aggregator that the proxy points to.

```
{/ prettier-ignore /}
solidity
function description() external view returns (string memory);
```

- RETURN: The description of the underlying aggregator.

#### getRoundData


Get data about a specific round, using the roundId.

```
{/ prettier-ignore /}
solidity
function getRoundData(
 uint80 roundId
) external view returns (uint80 roundId, int256 answer, uint256 startedAt,
 uint256 updatedAt, uint80 answeredInRound);
```

#### Parameters:

- roundId: The round ID

#### Return values:


- roundId: The round ID
- answer: The answer for this round
- startedAt: Timestamp of when the round started
- updatedAt: Timestamp of when the round was updated
- answeredInRound:  Deprecated - Previously used when answers could take multiple rounds to be computed

#### latestRoundData

Get the data from the latest round.

```
{/ prettier-ignore /}
solidity
function latestRoundData() external view
 returns (
 uint80 roundId,
 int256 answer,
 uint256 startedAt,
 uint256 updatedAt,
 uint80 answeredInRound
)
```

#### Return values:

- roundId: The round ID.
- answer: The data that this specific feed provides. Depending on the feed you selected, this answer provides asset prices, reserves, and other types of data.
- startedAt: Timestamp of when the round started.
- updatedAt: Timestamp of when the round was updated.
- answeredInRound:  Deprecated - Previously used when answers could take multiple rounds to be computed

version

The version representing the type of aggregator the proxy points to.

```
{/ prettier-ignore /}
solidity
function version() external view returns (uint256)
```

- RETURN: The version number.

#### AccessControlledOffchainAggregator

This is the contract for the aggregator. You can call functions on the aggregator directly, but it is a best practice to use the `AggregatorV3Interface` to run functions on the proxy instead so that changes to the aggregator do not affect your application. Read the aggregator contract only if you need functions that are not available in the proxy.

The aggregator contract has several variables and functions that might be useful for your application. Although aggregator contracts are similar for each data feed, some aggregators have different variables. Use the `typeAndVersion()` function on the aggregator to identify what type of aggregator it is and what version it is running.

Always check the contract source code and configuration to understand how specific data feeds operate. For example, the aggregator contract for BTC/USD on Arbitrum is different from the aggregators on other networks.

For examples of the contracts that are typically used in aggregator deployments, see the `libocr` repository on GitHub.

#### Variables and functions in AccessControlledOffchainAggregator

This contract imports `OffchainAggregator` and `SimpleReadAccessController`, which also include their own imports. The variables and functions lists include the publicly accessible items from these imported contracts.

A simple way to read the variables or functions is to get the ABI from a blockchain explorer and point the ABI to the aggregator address. To do this in Remix, follow the `Using the ABI with AtAddress` guide in the Remix documentation. As an example, you can find the ABI for the BTC/USD aggregator by viewing the contract code in Etherscan.

Variables:

Name	Description
-----	-----
-----	-----
-----	-----
-----	-----
LINK	The address for the LINK token contract on a specific network.
billingAccessController	The address for the billingAccessController, which limits access to the billing configuration for the aggregator.
checkEnabled	A boolean that indicates if access is limited to addresses on the internal access list.
maxAnswer	This value is no longer used on most Data Feeds.

Evaluate if your use case for Data Feeds requires a custom circuit breaker and

implement it to meet the needs of your application. See the Risk Mitigation page for more information. |

| minAnswer | This value is no longer used on most Data Feeds. Evaluate if your use case for Data Feeds requires a custom circuit breaker and implement it to meet the needs of your application. See the Risk Mitigation page for more information. |

| owner | The address that owns this aggregator contract. This controls which address can execute specific functions.

|

#### Functions:

Name	Description
-----	
-----	
decimals	Return the number of digits of precision for the stored answer. Answers are stored in fixed-point format.
description	Return a description for this data feed. This is different depending on which feed you select.
getAnswer	<Icon type="deprecated" /> Deprecated - Do not use this function.
getBilling configuration.	Retrieve the current billing configuration.
getRoundData	Get the full information for a specific aggregator round including the answer and update timestamps. Use this to get the full historical data for a round.
getTimestamp	<Icon type="deprecated" /> Deprecated - Do not use this function.
hasAccess	Check if an address has internal access.
latestAnswer	<Icon type="deprecated" /> Deprecated - Do not use this function.
latestConfigDetails	Return information about the current offchain reporting protocol configuration.
latestRound	<Icon type="deprecated" /> Deprecated - Do not use this function.
latestRoundData	Get the full information for the most recent round including the answer and update timestamps.
latestTimestamp	<Icon type="deprecated" /> Deprecated - Do not use this function.
latestTransmissionDetails	Get information about the most recent answer.
linkAvailableForPayment	Get the amount of LINK on this contract that is available to make payments to oracles. This value can be negative if there are outstanding payment obligations.
oracleObservationCount	Returns the number of observations that oracle is due to be reimbursed for.

```

| owedPayment | Returns how much LINK an oracle is
owed for its observations.
|
| requesterAccessController | Returns the address for the access controller
contract.
|
| transmitters | The oracle addresses that can report
answers to this aggregator.
|
| typeAndVersion | Returns the aggregator type and
version. Many aggregators are AccessControlledOffchainAggregator 3.0.0, but
there are other variants in production. The version is for the type of
aggregator, and different from the contract version. |
| validatorConfig | Returns the address and the gas limit
for the validator contract.
|
| version | Returns the contract version. This
is different from the typeAndVersion for the aggregator.
|

```

decimals

Return the number of digits of precision for the stored answer. Answers are stored in fixed-point format.

```

{/ prettier-ignore /}
solidity
function decimals() external view returns (uint8 decimalPlaces);

```

description

Return a description for this data feed. Usually this is an asset pair for a price feed.

```

{/ prettier-ignore /}
solidity
function description() public view override checkAccess returns (string memory)
{
 return super.description();
}

```

getAnswer

<Aside type="caution" title="This function is deprecated. Do not use this function."></Aside>

getBilling

Retrieve the current billing configuration.

```

{/ prettier-ignore /}
solidity
function getBilling()
 external
 view
 returns (
 uint32 maximumGasPrice,
 uint32 reasonableGasPrice,
 uint32 microLinkPerEth,
 uint32 linkGweiPerObservation,
)

```

```

 uint32 linkGweiPerTransmission
)
}
Billing memory billing = sbilling;
return (
 billing.maximumGasPrice,
 billing.reasonableGasPrice,
 billing.microLinkPerEth,
 billing.linkGweiPerObservation,
 billing.linkGweiPerTransmission
);
}

```

## getRoundData

Get the full information for a specific aggregator round including the answer and update timestamps. Use this to get the full historical data for a round.

```

{
 // prettier-ignore
 solidity
 function getRoundData(
 uint80 roundId
)
 public
 view
 override
 checkAccess
 returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound)
 {
 return super.getRoundData(roundId);
 }
}

```

## getTimestamp

<Aside type="caution" title="This function is deprecated. Do not use this function."></Aside>

## hasAccess

Check if an address has internal access.

```

{
 // prettier-ignore
 solidity
 function hasAccess(address user, bytes memory calldata) public view virtual
 override returns (bool) {
 return super.hasAccess(user, calldata) || user == tx.origin;
 }
}

```

## latestAnswer

<Aside type="caution" title="This function is deprecated. Do not use this function."></Aside>

## latestConfigDetails

Return information about the current offchain reporting protocol configuration.



```

{/ prettier-ignore /}
solidity
function latestConfigDetails() external view returns (uint32 configCount, uint32
blockNumber, bytes16 configDigest) {
 return (sconfigCount, slatestConfigBlockNumber, shotVars.latestConfigDigest);
}

```

## latestRound

<Aside type="caution" title="This function is deprecated. Do not use this function."></Aside>

## latestRoundData

Get the full information for the most recent round including the answer and update timestamps.

```

{/ prettier-ignore /}
solidity
function latestRoundData()
 public
 view
 override
 checkAccess
 returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt,
uint80 answeredInRound)
{
 return super.latestRoundData();
}

```

## latestTimestamp

<Aside type="caution" title="This function is deprecated. Do not use this function."></Aside>

## latestTransmissionDetails

Get information about the most recent answer.

```

{/ prettier-ignore /}
solidity
function latestTransmissionDetails()
 external
 view
 returns (bytes16 configDigest, uint32 epoch, uint8 round, int192 latestAnswer,
uint64 latestTimestamp)
{
 require(msg.sender == tx.origin, "Only callable by EOA");
 return (
 shotVars.latestConfigDigest,
 uint32(shotVars.latestEpochAndRound >> 8),
 uint8(shotVars.latestEpochAndRound),
 stransmissions[shotVars.latestAggregatorRoundId].answer,
 stransmissions[shotVars.latestAggregatorRoundId].timestamp
);
}

```

## linkAvailableForPayment

Get the amount of LINK on this contract that is available to make payments to oracles. This value can be negative if there are outstanding payment obligations.

```
{/ prettier-ignore /}
solidity
function linkAvailableForPayment() external view returns (int256
availableBalance) {
 // there are at most one billion LINK, so this cast is safe
 int256 balance = int256(LINK.balanceOf(address(this)));
 // according to the argument in the definition of totalLINKDue,
 // totalLINKDue is never greater than 2172, so this cast is safe
 int256 due = int256(totalLINKDue());
 // safe from overflow according to above sizes
 return int256(balance) - int256(due);
}
```

oracleObservationCount

Returns the number of observations that oracle is due to be reimbursed for.

```
{/ prettier-ignore /}
solidity
function oracleObservationCount(address signerOrTransmitter) external view
returns (uint16) {
 Oracle memory oracle = soracles[signerOrTransmitter];
 if (oracle.role == Role.Unset) {
 return 0;
 }
 return soracleObservationsCounts[oracle.index] - 1;
}
```

owedPayment

Returns how much LINK an oracle is owed for its observations.

```
{/ prettier-ignore /}
solidity
function owedPayment(address transmitter) public view returns (uint256) {
 Oracle memory oracle = soracles[transmitter];
 if (oracle.role == Role.Unset) {
 return 0;
 }
 Billing memory billing = sbilling;
 uint256 linkWeiAmount = uint256(soracleObservationsCounts[oracle.index] - 1)
 uint256(billing.linkGweiPerObservation)
 (1 gwei);
 linkWeiAmount += sgasReimbursementsLinkWei[oracle.index] - 1;
 return linkWeiAmount;
}
```

requesterAccessController

Returns the address for the access controller contract.

```
{/ prettier-ignore /}
solidity
```

```
function requesterAccessController() external view returns
(AccessControllerInterface) {
 return srequesterAccessController;
}
```

transmitters

The oracle addresses that can report answers to this aggregator.

```
{/ prettier-ignore /}
solidity
function transmitters() external view returns (address[] memory) {
 return stransmitters;
}
```

typeAndVersion

Returns the aggregator type and version. Many aggregators are AccessControlledOffchainAggregator 2.0.0, but there are other variants in production. The version is for the type of aggregator, and different from the contract version.

```
{/ prettier-ignore /}
solidity
function typeAndVersion() external pure virtual override returns (string memory)
{
 return "AccessControlledOffchainAggregator 2.0.0";
}
```

validatorConfig

Returns the address and the gas limit for the validator contract.

```
{/ prettier-ignore /}
solidity
function validatorConfig() external view returns (AggregatorValidatorInterface
validator, uint32 gasLimit) {
 ValidatorConfig memory vc = svalidatorConfig;
 return (vc.validator, vc.gasLimit);
}
```

version

Returns the contract version. This is different from the typeAndVersion for the aggregator.

```
{/ prettier-ignore /}
solidity
function version() external view returns (uint256);
```

# deprecating-feeds.mdx:

---

```
title: "Deprecation of Chainlink Data Feeds"
section: dataFeeds
metadata:
 description: "Deprecation of Chainlink Data Feeds"
date: Last Modified

```

```
import FeedPage from "@features/feeds/components/FeedPage.astro"
```

```
<FeedPage ecosystem="deprecating" />
```

```
developer-responsibilities.mdx:
```

```

section: dataFeeds
date: Last Modified
title: "Developer Responsibilities: Market Integrity and Application Code Risks"

```

```
import { ClickToZoom } from "@components"
```

Chainlink Data Feeds provide access to highly secure, reliable, and decentralized real-world data published onchain. The assets priced by Chainlink Data Feeds are subject to market conditions beyond the ability of Chainlink node operators to control, as such developers are responsible for ensuring that the operation and performance of Chainlink Data Feeds match expectations.

When integrating Chainlink Data Feeds, developers must understand that the performance of feeds is subject to risks associated with both market integrity and application code.

- Market Integrity Risks are those associated with external market conditions impacting price behavior and data quality in unanticipated ways. Developers are solely responsible for monitoring and mitigating any potential market integrity risks.

- Application Code Risks are those associated with the quality, reliability, and dependencies of the code on which an application operates. Developers are solely responsible for monitoring and mitigating any potential application code risks.

Please refer to this guide for additional information about market integrity risks and how developers can protect their applications.

## Developer Responsibilities

Developers are responsible for maintaining the security and user experience of their applications. They must also securely manage all interactions between their applications and third-party services.

In particular, developers implementing Chainlink Data Feeds in their code and applications are responsible for their application's market integrity and code risks that may cause unanticipated pricing data behavior. These are described below in more detail.

## Market Integrity Risks

Market conditions can impact the pricing behavior of assets in ways beyond the ability of Chainlink node operators to predict or control.

Market integrity risk factors can include but are not limited to, market manipulation such as Spoofing, Ramping, Bear Raids, Cross-Market Manipulation, Washtrading, and Frontrunning. All assets are susceptible to market risk, but in particular, assets with high market risk, such as those with low liquidity, are

the most vulnerable to market manipulation. Developers are solely responsible for accounting for such risk factors when integrating Chainlink Data Feeds into their applications. Developers should understand the market risks around the assets they intend their application to support before integrating associated Chainlink Data Feeds and inform their end users about applicable market risks.

Developers should reference the following additional information when implementing Chainlink Data Feeds:

- Data Feed Categories to evaluate market integrity risks associated with specific Chainlink Data Feeds Developers intend to integrate.
- Evaluating Data Source Risks to evaluate risk mitigation techniques associated with Chainlink Data Feeds broadly.

### Application Code Risks

Developers implementing Chainlink Data Feeds are solely responsible for instituting the requisite risk mitigation processes including, but not limited to, data quality checks, circuit breakers, and appropriate contingency logic for their use case.

- Code quality and reliability: Developers must execute code using Chainlink Data Feeds only if the code meets the quality and reliability requirements for their use case and application.
- Code and application audits: Developers are responsible for auditing their code and applications before deploying to production. Developers must determine the quality of any audits and ensure that they meet the requirements for their application.
- Code dependencies and imports: Developers are responsible for ensuring the quality, reliability, and security of any dependencies or imported packages that they use with Chainlink Data Feeds, and review and audit these dependencies and packages.

<ClickToZoom src="/images/developer-responsibilities.webp" />

# ens.mdx:

---

section: dataFeeds  
date: Last Modified  
title: "Using ENS with Data Feeds"  
excerpt: "Ethereum Name Service"  
---

```
import { EnsLookupForm } from "../../features/ens/components/EnsLookupForm.tsx"
import { EnsManualLookupForm } from
"../../features/ens/components/EnsManualLookupForm.tsx"
import { Aside, CodeSample } from "@components"
```

Lookup

<EnsLookupForm client:idle />

Manual Lookup

<EnsManualLookupForm client:idle />

Naming structure

Chainlink data feeds fall under the data.eth naming suffix. To obtain a specific feed address, prefix this with the assets in the feed, separated by a dash (-).

Pair	ENS Domain Name
ETH / USD	eth-usd.data.eth
BTC / USD	btc-usd.data.eth
...	...

## Subdomains

By default, the base name structure (eth-usd.data.eth) returns the proxy address for that feed. However, subdomains enable callers to retrieve other associated contract addresses, as shown in the following table.

Contract Addresses	Subdomain Prefix	Example
Proxy	proxy	proxy.eth-usd.data.eth
Underlying aggregator	aggregator	aggregator.eth-usd.data.eth
Proposed aggregator	proposed	proposed.eth-usd.data.eth

## Architecture

### Resolver

For each network, there is a single Chainlink resolver, which does not change. Its address can be obtained using the data.eth domain. This resolver manages the subdomains associated with data.eth.

Network	Resolver Address
Ethereum Mainnet	0x122eb74f9d0F1a5ed587F43D120C1c2BbDb9360B

### Listening for address changes

When a new aggregator is deployed for a specific feed, it is first proposed, and when accepted becomes the aggregator for that feed. During this process, the proposed and aggregator subdomains for that feed will change. With each change, the resolver emits an AddrChanged event, using the feed subdomain (for example: eth-usd.data.eth) as the indexed parameter.

Example: If you want to listen for when the aggregator of the ETH / USD feed changes, set up a listener to track the AddrChanged event on the resolver, using a filter like this: ethers.utils.namehash('aggregator.eth-usd.data.eth').

### Obtaining addresses

<Aside type="caution" title="Reverse Lookup">

Reverse lookup is not supported.

</Aside>

### Javascript

The example below uses Javascript Web3 library to interact with ENS. See the ENS documentation for the full list of languages and libraries that support ENS.

This example logs the address of the data feed on the Ethereum mainnet for ETH / USD prices.

<CodeSample src="samples/DataFeeds/ENSConsumer.js" />

## Solidity

In Solidity, the address of the ENS registry must be known. According to ENS documentation, this address is the same across Mainnet and testnet:

ENS registry address: 0x000000000000C2E074eC69A0dFb2997BA6C7d2e1e.

Also, instead of using readable string names like `eth-usd.data.eth`, resolvers accept bytes32 hash IDs for names. Hash IDs can be retrieved from this subgraph or via this npm package `eth-ens-namehash`.

"ETH / USD" hash:

```
0xf599f4cd075a34b92169cf57271da65a7a936c35e3f31e854447fbb3e7eb736d
```

```
<CodeSample src="samples/DataFeeds/ENSConsumer.sol" />
```

```
getting-started.mdx:
```

```

section: dataFeeds
date: Last Modified
title: "Consuming Data Feeds"
excerpt: "Smart Contracts and Chainlink"
whatsnext:
 {
 "See examples for how to read feeds onchain and offchain":
"/data-feeds/using-data-feeds",
 "Learn how to retrieve Historical Price Data": "/data-feeds/historical-
data",
 "Read the Data Feeds API Reference": "/data-feeds/api-reference",
 }
metadata:
 title: "Consuming Data Feeds"
 description: "Learn how to consume Chainlink Data Feeds in your smart
contracts."
 image: "/files/1a63254-link.png"

```

```
import { Aside, CodeSample } from "@components"
```

You can use Chainlink Data Feeds to connect your smart contracts to asset pricing data like the ETH / USD feed. These data feeds use data aggregated from many independent Chainlink node operators. Each price feed has an onchain address and functions that enable contracts to read pricing data from that address.

This guide shows you how to read Data Feeds and store the value onchain using Solidity. To learn how to read feeds offchain or use different languages, see the [Using Data Feeds on EVM Chains](#) guide. Alternatively, you can also learn how to use Data Feeds on Solana or StarkNet.

The code for reading Data Feeds on Ethereum or other EVM-compatible blockchains is the same for each chain and each Data Feed types. You choose different types of feeds for different uses, but the request and response format are the same. The answer decimal length and expected value ranges might change depending on what feed you use.

&lt;Aside type="caution" title="Using Data Feeds on L2 networks"&gt;

If you are using Chainlink Data Feeds on L2 networks like Arbitrum, Optimism, and Metis, you must also check the latest answer from the L2 Sequencer Uptime Feed to ensure that the data is accurate in the event of an L2 sequencer

outage. See the L2 Sequencer Uptime Feeds page to learn how to use Data Feeds on L2 networks.

</Aside>

Before you begin

If you are new to smart contract development, learn how to Deploy Your First Smart Contract before you start this guide.

Examine the sample contract

This example contract obtains the latest price answer from the BTC / USD feed on the Sepolia testnet, but you can modify it to read any of the different Types of Data Feeds.

<CodeSample src="samples/DataFeeds/DataConsumerV3.sol" />

The contract has the following components:

- The import line imports an interface named `AggregatorV3Interface`. Interfaces define functions without their implementation, which leaves inheriting contracts to define the actual implementation themselves. In this case, `AggregatorV3Interface` defines that all v3 Aggregators have the function `latestRoundData`. You can see the complete code for the `AggregatorV3Interface` on GitHub.
- The constructor() {} initializes an interface object named `dataFeed` that uses `AggregatorV3Interface` and connects specifically to a proxy aggregator contract that is already deployed at `0x1b44F3514812d835EB1BDB0acB33d3fA3351Ee43`. The interface allows your contract to run functions on that deployed aggregator contract.
- The `getChainlinkDataFeedLatestAnswer()` function calls your `dataFeed` object and runs the `latestRoundData()` function. When you deploy the contract, it initializes the `dataFeed` object to point to the aggregator at `0x1b44F3514812d835EB1BDB0acB33d3fA3351Ee43`, which is the proxy address for the Sepolia BTC / USD data feed. Your contract connects to that address and executes the function. The aggregator connects with several oracle nodes and aggregates the pricing data from those nodes. The response from the aggregator includes several variables, but `getChainlinkDataFeedLatestAnswer()` returns only the `answer` variable.

Compile, deploy, and run the contract

<Aside type="caution" title="Configure and fund your wallet">

If you have not already configured your MetaMask wallet and funded it with testnet ETH, follow the instructions in Deploy Your First Smart Contract to set that up. You can get testnet ETH at one of the available Sepolia faucets.

</Aside>

Deploy the `DataConsumerV3` smart contract on the Sepolia testnet.

1. Open the example contract in Remix. Remix opens and shows the contents of the smart contract.

```
{/ prettier-ignore /}
<div class="remix-callout">
 <a href="https://remix.ethereum.org/#url=https://docs.chain.link/samples/
DataFeeds/DataConsumerV3.sol">Open the contract in Remix
</div>
```



1. Because the code is already written, you can start the compile step. On the left side of Remix, click the Solidity Compiler tab to view the compiler settings.

!Screenshot showing the Compiler tab and its settings.

1. Use the default compiler settings. Click the Compile DataConsumerV3.sol button to compile the contract. Remix automatically detects the correct compiler version depending on the pragma that you specify in the contract. You can ignore warnings about unused local variables in this example.

!Screenshot of the Compile button.

1. On the Deploy tab, select the Injected Provider environment. This contract specifically requires Web3 because it connects with another contract on the blockchain. Running in a JavaScript VM will not work.

!Screenshot showing the Injected Provider environment selected.

1. Because the example contract has several imports, Remix might select another contract to deploy by default. In the Contract section, select the DataConsumerV3 contract to make sure that Remix deploys the correct contract.

!Screenshot showing DataConsumerV3 as the contract to deploy.

1. Click Deploy to deploy the contract to the Sepolia testnet. MetaMask opens and asks you to confirm payment for deploying the contract. Make sure MetaMask is set to the Sepolia network before you accept the transaction. Because these transactions are on the blockchain, they are not reversible.

!Screenshot of the Deploy button for DataConsumerV3.

1. In the MetaMask prompt, click Confirm to approve the transaction and spend your testnet ETH required to deploy the contract.

!Screenshot showing Metamask asking you to confirm the transaction.

1. After a few seconds, the transaction completes and your contract appears under the Deployed Contracts list in Remix. Click the contract dropdown to view its variables and functions.

!Remix Deployed Contracts Section

1. Click getChainlinkDataFeedLatestAnswer to show the latest answer from the aggregator contract. In this example, the answer is the latest price, which appears just below the button. The returned answer is an integer, so it is missing its decimal point. You can find the correct number of decimal places for this answer on the Price Feed addresses page by clicking the Show more details checkbox. The answer on the BTC / USD feed uses 8 decimal places, so an answer of 3030914000000 indicates a BTC / USD price of 30309.14. Each feed uses a different number of decimal places for answers.

!A screenshot showing the deployed contract and the latest answer.

# historical-data.mdx:

---

section: dataFeeds

date: Last Modified

title: "Getting Historical Data"

whatsnext: { "API Reference": "/data-feeds/api-reference" }

metadata:

title: "Getting Historical Data"

description: "How to use Chainlink Data Feeds to retrieve historical data in your smart contracts."  
---

```
import { priceFeedAddresses } from "@features/data"
import { HistoricalPrice } from "@features/feeds"
import { Aside, ClickToZoom, CodeSample, Icon } from "@components"
import button from "@chainlink/design-system/button.module.css"
```

The most common use case for Data Feeds is to Get the Latest Data from a feed. However, the `AggregatorV3Interface.sol` also has functions to retrieve data of a previous round IDs.

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

Name	Description
Deviation Threshold	Chainlink nodes are monitoring data offchain. The deviation of the real-world data beyond a certain interval triggers all the nodes to update.
Heartbeat Threshold	If the data values stay within the deviation parameters, it will only trigger an update every X minutes / hours.

You can find these parameters at [data.chain.link](https://data.chain.link) on an example like ETH / USD.

To learn how data feeds update, see the [Decentralized Data Model](#) page.

## Historical rounds

As shown in the decentralized model, the consumer contracts call the proxy contract, which abstracts the underlying aggregator contract. The main advantage is to enable upgrades of the aggregator without impacting the consumer contracts. That also means that historical data can be stored in different aggregators.

As shown in the following sequence diagram, to get historical data, call the `getRoundData` function and provide `roundId` as a parameter.

<ClickToZoom src="/images/data-feed/getRoundData-sequence.png" />

Note that `roundIds` have different meanings in proxy contracts and in aggregator contracts.

`roundId` in Aggregator (`aggregatorRoundId`)

Oracles provide periodic data updates to the aggregators. Data feeds are updated in rounds. Rounds are identified by their `roundId`, which increases with each new round. This increase may not be monotonic. Knowing the `roundId` of a previous round allows contracts to consume historical data.

The examples in this document name the aggregator `roundId` as `aggregatorRoundId` to differentiate it from the proxy `roundId`.

`roundId` in proxy

Because a proxy has references to current and all previous underlying aggregators, it needs a way to fetch data from the correct aggregator. The `roundId` is computed in the proxy contract as shown in the following example:

```
{/ prettier-ignore /}
```

```
solidity
return uint80(uint256(phase) << PHASEOFFSET | originalId);
```

where:

- phase is incremented each time the underlying aggregator implementation is updated. It is used as a key to find the aggregator address.
- originalId is the aggregator roundId. The ID starts at 1.

From the above formula, you can think of it as returning a large number containing the phase and the aggregator roundId.

<Aside type="note" title="Note">

The example formula above ensures that no matter how many times the underlying aggregator changes, the proxy roundId will always increase.

</Aside>

Example:

When you query historical data, it is important to know when you reach the end of the history of the underlying aggregator. As an example, if the latestRoundData function of the LINK / USD Price Feed on Ethereum Mainnet returns roundId = 92233720368547771158, you can use this value to compute the phaseId and aggregatorRoundId:

- phaseId = 92233720368547771158 >> 64: Right shifting an integer by 64 bits is equivalent to dividing it by  $2^{64}$ :  $\text{phaseId} = 92233720368547771158 / 2^{64} = 5$ . The current phase id is 5, which means that this proxy has had 5 underlying aggregators since its initial deployment.
- aggregatorRoundId = uint64(92233720368547771158): This retrieves the first 64 bits from the right. To calculate this offchain, you can use the following JavaScript example:

```
javascript
// First parse to BigInt to perform computation with big integers
const num = BigInt("92233720368547771158")
const num2 = BigInt("0xFFFFFFFFFFFFFFFF") // Largest 64bits integer

console.log(Number(num >> 64n)) // returns 5 (phaseId)
console.log(Number(num & num2)) // returns 13078 (aggregatorRoundId) . Use &
(AND bitwise operator) which sets each bit to 1 if both bits are 1
```

Using 13078 as the current aggregator's round, get its historical data by looping over the getRoundData function:

- Start from the first round: 92233720368547758081 (result of  $92233720368547771158 - 13078 + 1$ )
- Continue until the current round: 92233720368547771158

To get the historical data for previous aggregators, decrement the phaseId and start from round 1. For phase 4, get the starting roundId offchain using the following JavaScript example:

```
javascript
const phaseId = BigInt("4")
const aggregatorRoundId = BigInt("1")

roundId = (phaseId << 64n) | aggregatorRoundId // returns 73786976294838206465n
```

Loop over the getRoundData function. Start at 73786976294838206465 and increment

it until you get a revert. This means that you reached the last round for the underlying aggregator. The same process could be repeated for previous phaseIds (3,2,1).

<Aside type="caution" title="Looping onchain">

The examples showed how to loop offchain to fetch all historical data from a given proxy. You could also write a similar code onchain, but be aware that this could cause very high gas prices if a state is changed within the same function.

</Aside>

getRoundData return values

The getRoundData function returns the following values:

- roundId: The round in which the answer was updated
- answer: The answer reflects the data recorded for the specified round
- answeredInRound: <Icon type="deprecated" /> Deprecated - Previously used when answers could take multiple rounds to be computed
- startedAt: The timestamp when the round started
- updatedAt: The timestamp when the answer was computed

Solidity

<CodeSample src="samples/DataFeeds/HistoricalDataConsumer.sol" />

Javascript

<CodeSample src="samples/DataFeeds/HistoricalDataConsumer.js" />

```
<HistoricalPrice
 client:idle
 feedAddress={priceFeedAddresses.btc.usd.sepolia.address}
 roundId={priceFeedAddresses.btc.usd.sepolia.historicalRound}
 supportedChain="ETHEREUMSEPOLIA"
/>
```

Python

<CodeSample src="samples/DataFeeds/HistoricalDataConsumer.py" />

```
<a
 href="https://repl.it/@DwightLyle/GetHistoricalPriceWeb3PY"
 class={button.primary}
 style={{ "margin-top": "var(--space-2x)" }}
>
 {"Run this Python example"}

```

# index.mdx:

```

section: dataFeeds
date: Last Modified
title: "Chainlink Data Feeds"
isIndex: true
whatsnext:
 {
 "Follow the Data Feeds Getting Started guide to learn the basics": "/data-
feeds/getting-started",
 "Read the API reference for using Data Feeds": "/data-feeds/api-reference",
 "Find Rate Price Feed Addresses": "/data-feeds/price-feeds/addresses",
 "Find Rate Proof of Reserve Feed Addresses":
```

```

"/data-feeds/proof-of-reserve/addresses",
 "Find Rate and Volatility Feed Addresses":
"/data-feeds/rates-feeds/addresses",
 "Learn how to use Data Feeds on L2 networks": "/data-feeds/l2-sequencer-
feeds",
}
metadata:
 title: "Chainlink Data Feeds Documentation"
 description: "Add data to your smart contracts and applications."

```

```

import { Aside } from "@components"
import button from "@chainlink/design-system/button.module.css"

```

```

<Aside type="note" title="Talk to an expert">
 Contact us to talk to
an expert about integrating Chainlink
 Data Feeds with your applications.
</Aside>

```

Chainlink Data Feeds are the quickest way to connect your smart contracts to the real-world data such as asset prices, reserve balances, and L2 sequencer health.

If you already started a project and need to integrate Chainlink, you can add Chainlink to your existing project with the @chainlink/contracts NPM package.

## Types of data feeds

Data feeds provide many different types of data for your applications.

- Price Feeds
- Proof of Reserve Feeds
- Rate and Volatility Feeds
- L2 sequencer uptime feeds

## Price Feeds

Smart contracts often act in real-time on data such as prices of assets. This is especially true in DeFi.

For example, Synthetix uses Data Feeds to determine prices on their derivatives platform. Lending and borrowing platforms like AAVE use Data Feeds to ensure the total value of the collateral.

Data Feeds aggregate many data sources and publish them onchain using a combination of the Decentralized Data Model and Offchain Reporting.

To learn how to use Price Feeds, see the Price Feeds documentation.

See the Data Feeds Contract Addresses page for a list of available networks and addresses.

## Proof of Reserve Feeds

Proof of Reserves feeds provide the status of reserves for stablecoins, wrapped assets, and real world assets. Proof of Reserve Feeds operate similarly to Price Feeds, but provide answers in units of measurement such as ounces (oz) or number of tokens.

To learn more about Proof of Reserve Feeds, see the Proof of Reserve documentation.

See the Proof of Reserve Contract Addresses page for a list of available networks and addresses.

## Rate and Volatility Feeds

Several feeds provide interest rate curve data, APY data, and realized asset price volatility.

To learn more, see the [Rate and Volatility Feeds documentation](#).

See the [Rate and Volatility Contract Addresses](#) page for a list of available networks and addresses.

## L2 sequencer uptime feeds

L2 sequencer feeds track the last known status of the sequencer on an L2 network at a given point in time. This helps you prevent mass liquidations by providing a grace period to allow customers to react to these events.

To learn how to use L2 sequencer uptime feeds, see the [L2 Sequencer Uptime Feeds documentation](#).

## Components of a data feed

Data Feeds are an example of a decentralized oracle network and include the following components:

- Consumer: A consumer is an onchain or offchain application that uses Data Feeds. Consumer contracts use the `AggregatorV3Interface` to call functions on the proxy contract and retrieve information from the aggregator contract. For a complete list of functions available in the `AggregatorV3Interface`, see the [Data Feeds API Reference](#).
- Proxy contract: Proxy contracts are onchain proxies that point to the aggregator for a particular data feed. Using proxies enables the underlying aggregator to be upgraded without any service interruption to consuming contracts. Proxy contracts can vary from one data feed to another, but the `EACAggregatorProxy.sol` contract on Github is a common example.
- Aggregator contract: An aggregator is a contract that receives periodic data updates from the oracle network. Aggregators store aggregated data onchain so that consumers can retrieve it and act upon it within the same transaction. For a complete list of functions and variables available on most aggregator contracts, see the [Data Feeds API Reference](#).

To learn how to create a consumer contract that uses an existing data feed, read the [Using Data Feeds documentation](#).

## Reading proxy and aggregator configurations

Because the proxy and aggregator contracts are all onchain, you can see the current configuration by reading the variables through an ABI or using a blockchain explorer for your network. For example, you can see the BTC/USD proxy configuration on the Ethereum network using Etherscan.

If you read the BTC/USD proxy configuration, you can query all of the functions and variables that are publicly accessible for that contract including the aggregator address, `latestRoundData()` function, `latestAnswer` variable, owner address, `latestTimestamp` variable, and several others. To see descriptions for the proxy contract variables and functions, see the source code for your specific data feed on Etherscan.

The proxy contract points to an aggregator. This allows you to retrieve data through the proxy even if the aggregator is upgraded. If you view the aggregator address defined in the proxy configuration, you can see the aggregator and its configuration. For example, see the BTC/USD aggregator contract in Etherscan. This contract includes several variables and functions, including another `latestRoundData()`. To see descriptions for the aggregator variables and

functions, see the source code on GitHub or Etherscan.

You can call the `latestRoundData()` function directly on the aggregator, but it is a best practice to use the proxy instead so that changes to the aggregator do not affect your application. Similar to the proxy contract, the aggregator contract has a `latestAnswer` variable, owner address, `latestTimestamp` variable, and several others.

### Components of an aggregator

The aggregator contract has several variables and functions that might be useful for your application. Although aggregator contracts are similar for each data feed, some aggregators have different variables. Use the `typeAndVersion()` function on the aggregator to identify what type of aggregator it is and what version it is running.

Always check the contract source code and configuration to understand how specific data feeds operate. For example, the aggregator contract for BTC/USD on Arbitrum is different from the aggregators on other networks.

For examples of the contracts that are typically used in aggregator deployments, see the `libocr` repository on GitHub.

For a complete list of functions and variables available on most aggregator contracts, see the Data Feeds API Reference.

### Updates to proxy and aggregator contracts

To accommodate the dynamic nature of offchain environments, Chainlink Data Feeds are updated from time to time to add new features and capabilities as well as respond to externalities such as token migrations, protocol rebrands, extreme market events, and upstream issues with data or node operations.

These updates include changes to the aggregator configuration or a complete replacement of the aggregator that the proxy uses. If you consume data feeds through the proxy, your applications can continue to operate during these changes.

Proxy and aggregator contracts all have an owner address that has permission to change variables and functions. For example, if you read the BTC/USD proxy contract in Etherscan, you can see the owner address. This address is a multi-signature safe (multisig) that you can also inspect.

If you view the multisig contract in Etherscan using the Read as Proxy feature, you can see the full details of the multisig including the list of addresses that can sign and the number of signers required for the multisig to approve actions on any contracts that it owns.

The multisig-coordinated upgradability of Chainlink Data Feeds involves time-tested processes that balance collusion-resistance with the flexibility required to implement improvements and swiftly react to external conditions. The approach taken to upgradability will continue to evolve over time to meet user requirements.

### Monitoring data feeds

When you build applications and protocols that depend on data feeds, include monitoring and safeguards to protect against the negative impact of extreme market events, possible malicious activity on third-party venues or contracts, potential delays, and outages.

Create your own monitoring alerts based on deviations in the answers that data feeds provide. This will notify you when potential issues occur so you can respond to them.

Check the latest answer against reasonable limits

The data feed aggregator includes both `minAnswer` and `maxAnswer` values. On most data feeds, these values are no longer used and they do not stop your application from reading the most recent answer. For monitoring purposes, you must decide what limits are acceptable for your application.

Configure your application to detect when the reported answer is close to reaching reasonable minimum and maximum limits so it can alert you to potential market events. Separately, configure your application to detect and respond to extreme price volatility or prices that are outside of your acceptable limits.

Check the timestamp of the latest answer

Chainlink Data Feeds do not provide streaming data. Rather, the aggregator updates its `latestAnswer` when the value deviates beyond a specified threshold or when the heartbeat idle time has passed. You can find the heartbeat and deviation values for each data feed at `data.chain.link` or in the `Contract Addresses` lists.

Your application should track the `latestTimestamp` variable or use the `updatedAt` value from the `latestRoundData()` function to make sure that the latest answer is recent enough for your application to use it. If your application detects that the reported answer is not updated within the heartbeat or within time limits that you determine are acceptable for your application, pause operation or switch to an alternate operation mode while identifying the cause of the delay.

During periods of low volatility, the heartbeat triggers updates to the latest answer. Some heartbeats are configured to last several hours, so your application should check the timestamp and verify that the latest answer is recent enough for your application.

Users should build applications with the understanding that data feeds for wrapped or liquid staking assets might have different heartbeat and deviation thresholds than that of the underlying asset. Heartbeat and deviation thresholds can also differ for the same asset across different blockchains. Combining data from multiple feeds, even those with a common denominator, might result in a margin of error that users must account for in their risk mitigation practices.

To learn more about the heartbeat and deviation threshold, read the [Decentralized Data Model](#) page.

# l2-sequencer-feeds.mdx:

```

section: dataFeeds
date: Last Modified
title: "L2 Sequencer Uptime Feeds"

import { ClickToZoom, CodeSample } from "@components"
import { Aside } from "@components"
```

Optimistic rollup protocols move all execution off the layer 1 (L1) Ethereum chain, complete execution on a layer 2 (L2) chain, and return the results of the L2 execution back to the L1. These protocols have a sequencer that executes and rolls up the L2 transactions by batching multiple transactions into a single transaction.

If a sequencer becomes unavailable, it is impossible to access read/write APIs that consumers are using and applications on the L2 network will be down for most users without interacting directly through the L1 optimistic rollup



contracts. The L2 has not stopped, but it would be unfair to continue providing service on your applications when only a few users can use them.

To help your applications identify when the sequencer is unavailable, you can use a data feed that tracks the last known status of the sequencer at a given point in time. This helps you prevent mass liquidations by providing a grace period to allow customers to react to such an event.

## Available networks

You can find proxy addresses for the L2 sequencer feeds at the following addresses:

- Arbitrum:
  - Arbitrum mainnet: 0xFdB631F5EE196F0ed6FAa767959853A9F217697D
- Optimism:
  - Optimism mainnet: 0x371EAD81c9102C9BF4874A9075FFFFf170F2Ee389
- BASE:
  - BASE mainnet: 0xBCF85224fc0756B9Fa45aA7892530B47e10b6433
- Metis:
  - Andromeda mainnet: 0x58218ea7422255EBE94e56b504035a784b7AA204
- Scroll:
  - Scroll mainnet: 0x45c2b8C204568A03Dc7A2E32B71D67Fe97F908A9

## Arbitrum

The diagram below shows how these feeds update and how a consumer retrieves the status of the Arbitrum sequencer.

<ClickToZoom src="/images/data-feed/l2-diagram-arbitrum.webp" />

1. Chainlink nodes trigger an OCR round every 30s and update the sequencer status by calling the validate function in the ArbitrumValidator contract by calling it through the ValidatorProxy contract.
1. The ArbitrumValidator checks to see if the latest update is different from the previous update. If it detects a difference, it places a message in the Arbitrum inbox contract.
1. The inbox contract sends the message to the ArbitrumSequencerUptimeFeed contract. The message calls the updateStatus function in the ArbitrumSequencerUptimeFeed contract and updates the latest sequencer status to 0 if the sequencer is up and 1 if it is down. It also records the block timestamp to indicate when the message was sent from the L1 network.
1. A consumer contract on the L2 network can read these values from the ArbitrumUptimeFeedProxy contract, which reads values from the ArbitrumSequencerUptimeFeed contract.

## Handling Arbitrum outages

If the Arbitrum network becomes unavailable, the ArbitrumValidator contract continues to send messages to the L2 network through the delayed inbox on L1. This message stays there until the sequencer is back up again. When the sequencer comes back online after downtime, it processes all transactions from the delayed inbox before it accepts new transactions. The message that signals when the sequencer is down will be processed before any new messages with transactions that require the sequencer to be operational.

## Optimism, BASE, Metis, and Scroll

On Optimism, BASE, Metis, and Scroll, the sequencer's status is relayed from L1 to L2 where the consumer can retrieve it.

<ClickToZoom src="/images/data-feed/l2-diagram-optimism-metis.webp" />

On the L1 network:

1. A network of node operators runs the external adapter to post the latest sequencer status to the AggregatorProxy contract and relays the status to the Aggregator contract. The Aggregator contract calls the validate function in the OptimismValidator contract.

1. The OptimismValidator contract calls the sendMessage function in the L1CrossDomainMessenger contract. This message contains instructions to call the updateStatus(bool status, uint64 timestamp) function in the sequencer uptime feed deployed on the L2 network.

1. The L1CrossDomainMessenger contract calls the enqueue function to enqueue a new message to the CanonicalTransactionChain.

1. The Sequencer processes the transaction enqueued in the CanonicalTransactionChain contract to send it to the L2 contract.

On the L2 network:

1. The Sequencer posts the message to the L2CrossDomainMessenger contract.

1. The L2CrossDomainMessenger contract relays the message to the OptimismSequencerUptimeFeed contract.

1. The message relayed by the L2CrossDomainMessenger contains instructions to call updateStatus in the OptimismSequencerUptimeFeed contract.

1. Consumers can then read from the AggregatorProxy contract, which fetches the latest round data from the OptimismSequencerUptimeFeed contract.

Handling outages on Optimism, BASE, Metis, and Scroll

If the sequencer is down, messages cannot be transmitted from L1 to L2 and no L2 transactions are executed. Instead, messages are enqueued in the CanonicalTransactionChain on L1 and only processed in the order they arrived later when the sequencer comes back up. As long as the message from the validator on L1 is already enqueued in the CTC, the flag on the sequencer uptime feed on L2 will be guaranteed to be flipped prior to any subsequent transactions. The transaction that flips the flag on the uptime feed will be executed before transactions that were enqueued after it. This is further explained in the diagrams below.

When the Sequencer is down, all L2 transactions sent from the L1 network wait in the pending queue.

1. Transaction 3 contains Chainlink's transaction to set the status of the sequencer as being down on L2.

1. Transaction 4 is a transaction made by a consumer that is dependent on the sequencer status.

<ClickToZoom src="/images/data-feed/seq-down-1.webp" />

After the sequencer comes back up, it moves all transactions in the pending queue to the processed queue.

1. Transactions are processed in the order they arrived so Transaction 3 is processed before Transaction 4.

1. Because Transaction 3 happens before Transaction 4, Transaction 4 will read the status of the Sequencer as being down and responds accordingly.

<ClickToZoom src="/images/data-feed/seq-down-2.webp" />

Example code

This example code works on the Arbitrum, Optimism, and Metis networks. Create the consumer contract for sequencer uptime feeds similarly to the contracts that you use for other Chainlink Data Feeds. Configure the constructor using the following variables:

- Configure the `sequencerUptimeFeed` object with the sequencer uptime feed proxy address for your L2 network.
- Configure the `dataFeed` object with one of the Data Feed proxy addresses that are available for your network.

```
<CodeSample src="samples/DataFeeds/DataConsumerWithSequencerCheck.sol" />
```

The `sequencerUptimeFeed` object returns the following values:

- `answer`: A variable with a value of either 0 or 1
  - 0: The sequencer is up
  - 1: The sequencer is down
- `startedAt`: This timestamp indicates when the sequencer feed changed status. When the sequencer comes back up after an outage, wait for the `GRACEPERIODTIME` to pass before accepting answers from the data feed. Subtract `startedAt` from `block.timestamp` and revert the request if the result is less than the `GRACEPERIODTIME`.
  - The `startedAt` variable returns 0 only on Arbitrum when the Sequencer Uptime contract is not yet initialized. For L2 chains other than Arbitrum, `startedAt` is set to `block.timestamp` on construction and `startedAt` is never 0. After the feed begins rounds, the `startedAt` timestamp will always indicate when the sequencer feed last changed status.

If the sequencer is up and the `GRACEPERIODTIME` has passed, the function retrieves the latest answer from the data feed using the `dataFeed` object.

# selecting-data-feeds.mdx:

```

section: dataFeeds
date: Last Modified
title: "Selecting Quality Data Feeds"
excerpt: "Learn how to assess data feeds that you use in your smart contracts."

```

```
import { Aside } from "@components"
```

When you design your applications, consider the quality of the data that you use in your smart contracts. Ultimately you are responsible for identifying and assessing the accuracy, availability, and quality of data that you choose to consume via the Chainlink Network. Note that all feeds contain some inherent risk. Read the Risk Mitigation and Evaluating Data Sources sections when making design decisions. Chainlink lists decentralized data feeds in the documentation to help developers build new applications integrated with data.

## Data Feed Categories

This categorization is put in place to inform users about the intended use cases of feeds and help highlight some of the inherent market integrity risks surrounding the data quality of these feeds.

All feeds published on `docs.chain.link` are monitored and maintained to the same levels and standards. Each feed goes through a rigorous assessment process when implemented. The assessment criteria can vary depending on the product type of feed being deployed or change over time as the understanding of market integrity risks evolves.

Market price feeds incorporate three layers of aggregation at the data source,

node operator, and oracle network layers, providing industry-standard security and reliability on the price data they reference. To learn more about the three layers of data aggregation, see the blog post about [Data Aggregation in Chainlink Price Feeds](#). Additional information about how Chainlink Data Feeds are secured can be seen in the blog post about [How Chainlink Price Feeds Secure the DeFi Ecosystem](#).

Data feeds are grouped into the following categories based on the level of market integrity risk from lowest to highest:

- 🟢 Low Market Risk
- 🟡 Medium Market Risk
- 🔴 High Market Risk
- 🟡 New Token Feeds
- 🟡 Custom Feeds
- ⚠️ Deprecating

<Aside type="note">

For important updates regarding the use of Chainlink Data Feeds, users should join the official Chainlink Discord and subscribe to the #data-feeds channel.

</Aside>

#### 🟢 Low Market Risk Feeds

These are data feeds that follow a standardized data feeds workflow to report market prices for an asset pair. Chainlink node operators each query several sources for the market price and aggregate the estimates provided by those sources.

Low Market Risk feeds have the following characteristics:

- Highly resilient to disruption
- Leverage many data sources
- High volumes across a large number of markets enable consistent price discovery

While market risk may be low, other risks might still exist based on your use case, the blockchain on which the feed is deployed, and the conditions on that chain.

#### 🟡 Medium Market Risk Feeds

These feeds also follow a standardized data feeds workflow to report market prices for an asset pair. The pair in question may have features that make it more challenging to reliably price, or potentially subject it to volatility which may pose a risk in some use cases. While the architecture of these feeds is resilient and distributed, these feeds carry additional market risk.

Types of market risk that may lead to a feed being categorized as Medium Market Risk include:

- Lower or inconsistent asset volume may result in periods of low liquidity in the market for such assets. This, in turn, can lead to volatile price movements
- A spread between the price for this asset on different trading venues or liquidity pools.
- Market Concentration Risk: If the volume for a given asset is excessively concentrated on a single exchange, that trading venue could become a single point of failure for the feed.
- Cross-Rate Risk: The base asset trades in large volumes against assets that are not pegged to the quote asset. As a result, the price of this specific asset pair may fluctuate even if the underlying asset is not being traded.
- The asset is going through a significant market event such as a token or liquidity migration.

- The asset has a high spread between data providers, the root cause of which is often one of the above factors.

#### High Market Risk Feeds

These feeds also follow a standardized data feeds workflow to report market prices for an asset pair. However, the pair in question often exhibits a heightened degree of some of the risk factors outlined under "Medium Market Risk", or a separate risk that makes the market price subject to uncertainty or volatility. In using a High Market Risk data feed you acknowledge that you understand the risks associated with such a feed and that you are solely responsible for monitoring and mitigating such risks.

Types of market risk that may lead to a feed being categorized as High Market Risk include:

- The asset is going through a significant market event such as a hack, bridge failure, or a delisting from a major exchange.
- The asset or project is being deprecated in the market.
- Volumes have dropped to extremely low levels.

#### New Token Feeds

When a token is newly launched, the historical data required to implement a rigorous risk assessment framework that would allow the categorization of a market data feed for that token as low, medium, or high market risk is unavailable. Consistent price discovery may involve an indeterminate amount of time. Users must understand the additional market and volatility risks inherent with such assets. Users of new token feeds are responsible for independently verifying the liquidity and stability of the assets priced by the feeds that they use.

At the end of a probationary period, the status of new token feeds may be adjusted to high/medium/low market risk or in rare cases be deprecated entirely.

#### Custom Feeds

Custom Feeds are built to serve a specific use case and might not be suitable for general use or your use case's risk parameters. Users must evaluate the properties of a feed to make sure it aligns with their intended use case. Contact the Chainlink Labs team if you want more detail on any specific feeds in this category.

Custom feeds have the following categories and compositions:

- Onchain single source feeds: These feeds take their data from an onchain source, however, the feed has only a single data provider currently supporting the feed.
- Onchain Proof of Reserve Feeds: Chainlink Proof of Reserve uses a large decentralized collection of security-reviewed and Sybil-resistant node operators to acquire and verify reserve data. In this use case, reserves reside onchain.
- Exchange Rate Feeds: These feeds read an exchange rate from an external contract onchain that is designed to allow conversion from one token to another. Chainlink does not own or control these contracts in any way. They are not equivalent to market price feeds.
- Technical Feeds: Feeds within this category measure a particular technical metric from a specified blockchain. For example, Fast Gas or Block Difficulty.
- Total Value Locked Feeds: These feeds measure the total value locked in a particular protocol.
- Custom Index Feeds: An index calculates a function of the values for multiple underlying assets. The function is specific to that index and is typically calculated by node operators following an agreed formula.
- Offchain Single Source Feeds: Some data providers use a single data source, which might be necessary if only one source exists offchain for a specific type of data.

- Offchain Proof of Reserve Feeds: Chainlink Proof of Reserve uses a large decentralized collection of security-reviewed and Sybil-resistant node operators to acquire and verify reserve data. In this use case, reserves reside offchain.
- LP Token Feeds: These feeds use a decentralized feed for the underlying asset as well as calculations to value the liquidity pool (LP) tokens.
- Wrapped Calculated Feeds: These feeds are typically pegged 1:1 to the underlying token or asset. Under normal market conditions, these feeds track their underlying value accurately. However, given that the price is a derivative formed from a calculated method, the derivative asset may not always precisely track the value of the underlying token or asset precisely.

If you plan on using one of these feeds and would like to get a more detailed understanding, contact the Chainlink Labs team.

## â Deprecating

These feeds are being deprecated. To find the deprecation dates for specific feeds, see the Feeds Scheduled For Deprecation page.

## Data Feed Shutdown Policy

Data feeds managed by Chainlink Labs will be considered for deprecation if they pose a risk to the Chainlink Community and broader ecosystem, if the asset or assets on the feed have significantly deteriorated and no longer meet our Quality Assurance standards, or if the data feed has become economically unsustainable to support.

Known users of these feeds will be contacted directly about the feeds' deprecation. Notifications will also be posted on the Feeds Scheduled For Deprecation page and on our Discord channel with two weeks of notice before they are shut down.

## Market hours

In addition to categories, be aware that markets for several assets are actively traded only during certain hours. Listed data feeds include an attribute describing their market hours. Chainlink Labs recommends using these feeds only during their specified hours:

Feed	Hours
-----	
-----	
-----	
-----	
Crypto	24/7/365 - No market close.
USEquities	Standard US equity market hours: 09:30 - 16:00 ET M-F excluding US equity market holidays.
UKETF	Standard UK equity market hours: 08:00 - 16:30 UK time M-F excluding UK equity market holidays.
Forex	18:00 ET Sunday to 17:00 ET Friday.   The feeds also follow the global Forex market Christmas and New Year's Day holiday schedule. Many non-G12 currencies primarily trade during local market hours. It is recommended to use those feeds only during local trading hours.
PreciousMetals	18:00 ET Sunday to 17:00 ET Friday with a one- hour break Monday through Thursday from 17:00 to 18:00.   The feeds also follow the global Forex market holiday schedule for Christmas and New Year's Day.
NYMEX (US OIL)	18:00 ET Sunday to 17:00 ET Friday, with a one- hour break Monday through Thursday from 17:00 to 18:00.   The feed also

follows the NYMEX market holiday schedule.

|  
| COMEX (Non precious metals) | 18:00 ET Sunday to 17:00 ET Friday with a one-hour break Monday through Thursday from 17:00 to 18:00. <br/> The feed also follows the COMEX market holiday schedule.

|  
| CBOT (Agricultural) | Monday - Thursday 00:00-08:45, 09:30-00:00 and Friday 00:00-08:45, 09:30-14:00 ET, excluding CBOT market holidays.

|  
| MIETF | Standard Milan equity market hours. 09:00-17:30 CET time M-F, excluding Milan equity market holidays.

|  
| XETRAETF | Deutsche Börse Xetra equity market hours. 09:00-17:30 CET time M-F, excluding Xetra equity market holidays.

## Risk Mitigation

As a development best practice, design your systems and smart contracts to be resilient and mitigate risk to your protocol and your users. Ensure that your systems can tolerate known and unknown exceptions that might occur. Some examples include but are not limited to volatile market conditions, the degraded performance of infrastructure, chains, or networks, and any other upstream outage related to data providers or node operators. You bear responsibility for any manner in which you use the Chainlink Network, its software, and documentation.

To help you prepare for unforeseen market events, you should take additional steps to protect your application or protocol regardless of the market risk categorization of the Data Feeds your application consumes. The below tooling is put in place to mitigate extreme market events, possible malicious activity on third-party venues or contracts, potential delays, performance degradation, and outages.

Below are some examples of tooling that Chainlink users have put in place:

- Circuit breakers: In the case of an extreme price event, the contract would pause operations for a limited period of time. Chainlink Automation is able to monitor data feeds to identify unexpected events. If an event were to occur, the Automation network can send an onchain transaction to pause or halt contract functionality.
- Contract update delays: Contracts would not update until the protocol had received a recent fresh input from the data feed.
- Manual kill switch: If a vulnerability or bug is discovered in one of the upstream contracts, the user can manually cease operation and temporarily sever the connection to the data feed.
- Monitoring: Some users create their own monitoring alerts based on deviations in the data feeds that they are using.
- Soak testing: Users are strongly advised to thoroughly test price feed integrations and incorporate a soak period prior to providing access to end users or securing value.

For more detailed information about some of these examples, see the Monitoring data feeds documentation.

For important updates regarding the use of Chainlink Price Feeds, users should join the official Chainlink Discord and subscribe to the data-feeds-user-notifications channel.

## Chainlink Community Deployments

Chainlink technology is used by many within the blockchain community to support their use cases. Deployments built and run by community members are not tracked in the Chainlink documentation. Chainlink's community is continuously growing, and they play a vital role in developing the ecosystem, so the software and

tooling are developed for anyone to use. Users have a wide variety of options for choosing how to deliver data onchain. They can deploy Chainlink nodes themselves or via the extensive network of node operators that offer services and access one of the community-managed oracle networks that support the supply of various types of data onchain. Chainlink Labs does not take responsibility for the use of Chainlink node software.

It is always recommended that you conduct a thorough analysis of your requirements and carry out appropriate due diligence on any partners you wish to use with your project.

The Chainlink Labs team does not monitor community deployments and users should use best practices in observability, monitoring, and risk mitigation as appropriate for your application's stage of development and use case.

As your usage of data feeds evolves and requirements for higher availability and greater security increases, such as securing substantive value, the reliability properties of your data feed will become crucial. Contact Chainlink Labs team for services to ensure deployments meet the highest levels of availability and security.

High Risk: Forked, modified, or custom software:

As Chainlink is open source, independent forks and modifications may exist. Chainlink Labs and development teams are not involved in these and do not track or maintain visibility on them. Chainlink Labs is not responsible for updates, enhancements, or bug fixes for these versions, and Chainlink Labs does not monitor them. Their use might pose risks that can do harm to your project. Users are responsible for thoroughly vetting and validating such deployments and determining their suitability.

## Evaluating Data Sources and Risks

If your smart contracts use data feeds, assess those data feeds for the following characteristics:

- Liquidity and its Distribution
- Single Source Data Providers
- Crypto and Blockchain Actions
- Market Failures Resulting from Extreme Events
- Periods of High Network Congestion
- Unknown and Known Users
- DEX volumes

## Liquidity and its Distribution

If your smart contract relies on pricing data for a specific asset, make sure that the asset has a sufficiently healthy level of liquidity in the market to avoid price and market manipulation. Assets with low liquidity or volume can be volatile, which might negatively impact your application and its users. Malicious actors might try to exploit volatility or periods of reduced trading activity to take advantage of the logic in a smart contract and cause it to execute in a way that you did not intend.

Some data feeds obtain their pricing data from individual exchanges rather than from aggregated price tracking services that gather their data from multiple exchanges. These are marked as such in the docs page for that feed. Assess the liquidity and reliability of that specific exchange.

Liquidity migrations occur when a project moves its tokens from one liquidity provider (such as a DEX, a CEX, or a new DeFi application) to another. When liquidity migrations occur, it can result in low liquidity in the original pool, making the asset susceptible to market manipulation. If your project is considering a liquidity migration, you should coordinate with relevant



stakeholders, including liquidity providers, exchanges, oracle node operators, Data Feed providers, and users, to ensure prices are accurately reported throughout the migration.

Feeds for assets with low market liquidity or volume where data providers exhibit an abnormal price spread may, on occasion, see a price oscillate between two or more price points within regular intervals. To mitigate risk associated with such price oscillation, users must regularly monitor & assess the quality of an asset's liquidity. Similarly, assets with low market liquidity may experience abnormal or volatile price movements due to erroneous trades.

Design and test your contracts to handle price spikes and implement risk management measures to protect your assets. For example, create mock tests that return various oracle responses.

### Single Source Data Providers

Some data providers use a single data source, which might be necessary if only one source exists onchain or offchain for a specific type of data. Evaluate data providers to make sure they provide high-quality data that your smart contracts can rely on. Any error or omission in the provider's data might negatively impact your application and its users.

### Crypto and Blockchain Actions

Price data quality is subject to crypto actions by the crypto and blockchain project teams. Crypto actions are similar to corporate actions but are specific to cryptocurrency and blockchain projects, such as token renaming, token swaps, redenominations, splits, reverse splits, network upgrades, and other migrations that teams or communities who govern the blockchain or token might undertake.

Sustaining data quality is dependent on data sources implementing the necessary adjustments related to such actions. For example, when a project upgrades to a new version of their token, this results in a token migration. When token migrations occur, they require building a new Data Feed to ensure that the token price is accurately reported. Similarly, actions by blockchain project teams or communities, such as forks or upgrades to the network, may require new Data Feeds to ensure continuity and data quality. When considering a token migration, fork, network upgrade, or other crypto action, projects should proactively reach out to relevant stakeholders to ensure the asset price is accurately reported throughout the process.

### Market Failures Resulting from Extreme Events

Users are strongly advised to set up monitoring and alerts in the event of unexpected market failures. Black swan events, hacks, coordinated attacks, or extreme market conditions may trigger unanticipated outcomes such as liquidity pools becoming unbalanced, unexpected re-weighting of indices, abnormal behavior by centralized or decentralized exchanges, or the de-pegging of synthetic assets, stablecoins, and currencies from their intended exchange rates.

Circuit breakers can be created using Chainlink Automation. Circuit breakers are safety measures that monitor data feeds for unexpected scenarios such as stale prices, drastic price changes, or prices approaching a predetermined min/max threshold. If an unexpected scenario occurs, the circuit breaker can send an onchain transaction to pause or halt contract functionality.

### Periods of High Network Congestion

Data Feed performance relies on the chains they are deployed on. Periods of high network congestion or network downtime might impact the frequency of Chainlink Data Feeds. It is advised that you configure your applications to detect such chain performance or reliability issues and to respond appropriately.

## Unknown and Known Users

Routine maintenance is carried out on Chainlink Data Feeds, including decommissioning, on an ad-hoc basis. These maintenance periods might require users to take action in order to maintain business continuity.

Notifications are sent to inform known users regarding such occurrences, and it is strongly encouraged for all users, including those users utilizing data feeds for offchain purposes, to provide their contact information before utilizing data feeds. Without providing contact information, users will be unable to receive notifications regarding important Data Feed updates.

If you are using Data Feeds but have not provided your contact information, you can do so [here](#). Users that fail to provide notification information do so at their own risk.

## DEX Volumes

Assets with a significant market presence on decentralized exchanges (DEXs) face distinct risks related to unique market structure. The market integrity can be compromised by flash loan-funded attacks, volume shifts to different onchain or offchain exchanges, or a well-capitalized actor temporarily manipulating the price on that exchange. Additionally, DEX trades can result in slippage due to liquidity migrations and trade size. The likelihood of high-slippage trades being accurately reflected in market prices depends on the trading patterns of the asset.

Generally, a lower risk of deviant trades impacting aggregated prices is associated with assets having multiple DEX pools with healthy volumes and consistent trading activity across different time windows.

## Evaluating Wrapped or Bridged Assets

### Assessing how to Price Wrapped or Bridged Assets

When assessing a Chainlink Data Feed for a wrapped or bridged asset such as WBTC, users should evaluate the tradeoffs between using a Data Feed specifically built for the wrapped or bridged asset or a Data Feed built for the underlying asset.

Decisions should be made on a case-by-case basis considering the liquidity, depth, and trading volatility of the underlying asset compared to its derivative. In addition, users must consider the security mechanism that is designed to keep the wrapped or bridged asset coupled to its underlying asset. Review these parameters regularly as asset dynamics continuously evolve.

### Extreme Events Causing Price Deviations in Wrapped or Bridged Assets

Chainlink Data Feeds are designed to provide the market-wide price of various assets, as determined by a volume-weighted average across a wide range of exchanges. On blockchain networks where assets are wrapped and/or bridged from another environment using a cross-chain token bridge, Chainlink Data Feeds built for the underlying asset will continue to report the market-wide price of the underlying asset as opposed to the price of the wrapped/bridged asset. This methodology reduces risks around market manipulation because wrapped/bridged tokens are often less liquid than the underlying asset.

However, users should be aware that certain extreme events may result in price deviations between the wrapped/bridged asset and its underlying counterpart. For example, the exploitation or hack of a cross-chain token bridge may cause a collapse in demand for a particular wrapped asset. As such, users should construct their applications with safeguards, such as circuit breakers to proactively pause functionality to mitigate risk during such scenarios. Circuit breakers can be created using Chainlink Automation to monitor data feeds for

unexpected scenarios.

An additional mechanism for securing a protocol utilizing wrapped assets is by incorporating Chainlink Proof of Reserve. Chainlink Proof of Reserve enables the real-time reserve monitoring of offchain and cross-chain assets, including those that have been wrapped/bridged. By comparing the wrapped token's supply against a Chainlink Proof of Reserve feed, protocols can ensure that these assets are properly collateralized at all times.

#### Front Running Risk

Front running (when a third party benefits from prior access to information about a transaction) is a known risk inherent to specific blockchain applications. Chainlink Data Feeds are optimized to prioritize high levels of data quality and reliability over latency.

To mitigate the risk associated with front running, users building highly latency-dependent applications should assess whether the configuration of data feeds meets their needed specifications for speed and frequency. Chainlink Data Streams serve as an alternative solution to Data Feeds for latency-sensitive applications, providing low-latency delivery of market data offchain that can be verified onchain while mitigating front running.

#### Exchange Rate Feeds

The architecture of exchange rate feeds differs from that of standard market rate Chainlink Price Feeds.

Market rate feeds (e.g., Chainlink Price Feeds) deliver price updates based on the volume-based aggregated market price of a specific asset. Price data is aggregated from across multiple sources, including centralized and decentralized exchanges, to provide an accurate representation of an asset's market-wide price.

Exchange rate feeds are tied to specific protocols or ecosystems and report the internal redemption rates for an asset (i.e., the value/rate at which an asset can be redeemed or exchanged within that protocol's ecosystem). This data is sourced directly from a specified smart contract on a source chain and relayed to a destination chain.

Exchange rate feeds are useful in circumstances such as:

- Pricing yield-bearing assets: Multiplying a yield-bearing asset's exchange rate by the underlying asset's market rate can be used to calculate the yield-bearing asset's current price. This methodology can reduce certain pricing volatility risks associated with lower-liquidity yield-bearing assets.
- Enabling cross-chain staking: For example, liquid staking tokens (LSTs) or liquid restaking tokens (LRTs) can be minted at the exchange rate on a layer 2, while Chainlink CCIP transfers the underlying asset to a layer 1 for staking.
- Improving liquidity pool performance for yield-bearing assets: Exchange rate feeds can be utilized to programmatically adjust swap curves to maximize liquidity efficiency.

<Aside type="note">

Users must be aware that both market rate and exchange rate pricing methodologies have unique risk considerations and mitigation strategies that vary based on an asset's type and liquidity profile. Users are responsible for decisions relating to feed selection. To learn more about how best to leverage Chainlink Data Feeds for your project, contact us [here](#).

</Aside>

#### ETF and Forex feeds

When you use Data Feeds for ETFs or Foreign Exchange (Forex) data, be aware of the following best practices:

- Offchain equity and ETF assets are traded only during standard market hours. Do not use these feeds outside those windows.
- Assets on the Forex (Foreign Exchange) markets are traded only during defined market hours. Additionally, some currencies might trade only during local banking hours. Do not use Forex feeds outside market hours for the specific currency.
- UK ETF price feed answers are 15 minutes delayed from their original published source. Assets are traded only during standard market hours. Do not use these feeds outside their specified hours.

# using-data-feeds.mdx:

```

section: dataFeeds
date: Last Modified
title: "Using Data Feeds on EVM Chains"
whatsnext:
 { "Historical Price Data": "/data-feeds/historical-data", "Data Feeds API
Reference": "/data-feeds/api-reference" }

import { LatestPrice } from "@features/feeds"
import { priceFeedAddresses } from "@features/data"
import { Aside, CodeSample } from "@components"
import { Tabs } from "@components/Tabs"
import button from "@chainlink/design-system/button.module.css"
```

The code for reading Data Feeds is the same across all EVM-compatible blockchains and Data Feed types. You choose different types of feeds for different uses, but the request and response format are the same. To read a feed, specify the following variables:

- RPC endpoint URL: This determines which network that your smart contracts will run on. You can use a node provider service or point to your own client. If you are using a Web3 wallet, it is already configured with the RPC endpoints for several networks and the Remix IDE will automatically detect them for you.
- LINK token contract address: The address for the LINK token contract is different for each network. You can find the full list of addresses for all supported networks on the [LINK Token Contracts](#) page.
- Feed contract address: This determines which data feed your smart contract will read. Contract addresses are different for each network. You can find the available contract addresses on the following pages:
  - [Price Feed Addresses](#)
  - [Proof of Reserve Feed Addresses](#)

The examples in this document indicate these variables, but you can modify the examples to work on different networks and read different feeds.

This guide shows example code that reads data feeds using the following languages:

- Onchain consumer contracts:
  - Solidity
  - Vyper
- Offchain reads using Web3 packages:
  - Javascript with web3.js
  - Python with Web3.py
  - Golang with go-ethereum

## Reading data feeds onchain

These code examples demonstrate how to deploy a consumer contract onchain that reads a data feed and stores the value.

### Solidity

To consume price data, your smart contract should reference `AggregatorV3Interface`, which defines the external functions implemented by Data Feeds.

```
<CodeSample src="samples/DataFeeds/DataConsumerV3.sol" />
```

The `latestRoundData` function returns five values representing information about the latest price data. See the Data Feeds API Reference for more details.

### Vyper

To consume price data, your smart contract should import `AggregatorV3Interface` which defines the external functions implemented by Data Feeds. You can find it [here](#).

You can find a `PriceConsumer` example [here](#). Read the `apeworx-starter-kit` README to learn how to run the example.

## Reading data feeds offchain

These code examples demonstrate how to read data feeds directly off chain using Web3 packages for each language.

### Javascript

This example uses `web3.js` to retrieve feed data from the BTC / USD feed on the Sepolia testnet.

```
<Tabs client:visible>
 <Fragment slot="tab.1">web3.js</Fragment>
 <Fragment slot="tab.2">ethers.js</Fragment>
 <Fragment slot="panel.1">
 <CodeSample src="samples/DataFeeds/PriceConsumerV3.js" />
 </Fragment>
 <Fragment slot="panel.2">
 <CodeSample src="samples/DataFeeds/PriceConsumerV3Ethers.js" />
 </Fragment>
</Tabs>
```

```
<LatestPrice client:idle
feedAddress={priceFeedAddresses.btc.usd.sepolia.address}
supportedChain="ETHEREUMSEPOLIA" />
```

### Python

This example uses `Web3.py` to retrieve feed data from the BTC / USD feed on the Sepolia testnet.

```
<CodeSample src="samples/DataFeeds/PriceConsumerV3.py" />
<a href="https://repl.it/@DwightLyle/GetLatestPriceWeb3PY"
class={button.primary}>
 {"Run this Python example"}

```

### Golang

You can find an example with all the source files [here](#). This example uses `go-ethereum` to retrieve feed data from the BTC / USD feed on the Sepolia testnet.

To learn how to run the example, see the README.

<Aside type="note" title="New Feed Registry">

You can use the Feed Registry to reference data feed assets by name or currency

identifier instead of by pair/proxy address.

</Aside>

Getting a different price denomination

Chainlink Data Feeds can be used in combination to derive denominated price pairs in other currencies.

If you require a denomination other than what is provided, you can use two data feeds to derive the pair that you need. For example, if you needed a BTC / EUR price, you could take the BTC / USD feed and the EUR / USD feed and derive BTC / EUR using division.

!Request Model Diagram

<Aside type="caution" title="Important">

If your contracts require Solidity versions that are  $\geq 0.6.0$   $< 0.8.0$ , use OpenZeppelin's SafeMath version 3.4.

</Aside>

<CodeSample src="samples/DataFeeds/PriceConverter.sol" />

More aggregator functions

Getting the latest price is not the only data that aggregators can retrieve. You can also retrieve historical price data. To learn more, see the Historical Price Data page.

To understand different use cases for Chainlink Price Feeds, refer to Other Tutorials.

# feed-registry-functions.mdx:

---

section: dataFeeds

date: Last Modified

title: "Feed Registry API Reference"

metadata:

title: "Feed Registry API Reference"

description: "Chainlink Feed Registry Functions"

---

import { Icon } from "@components"

This guide outlines the functions which can be used with Chainlink's Feed Registry. You can learn more about the feed registry [here](#).

Functions

Name	Description
decimals	The number of decimals in the response.
description	The description of the aggregator that the proxy points to.

```

| getRoundData | Get data from a specific round.
|
| latestRoundData | Get data from the latest round.
|
| version | The version representing the type of
aggregator the proxy points to.
| getFeed | Returns the primary aggregator address of a
base / quote pair.
| getPhaseFeed | Returns the aggregator address of a base / quote
pair at a specified phase.
| isFeedEnabled | Returns true if an aggregator is enabled as primary
on the registry.
| getPhase | Returns the raw starting and ending aggregator
round ids of a base / quote pair.
| getRoundFeed | Returns the underlying aggregator address of a base
/ quote pair at a specified round.
| getPhaseRange | Returns the starting and ending round ids of a
base / quote pair at a specified phase.
| getPreviousRoundId | Returns the previous round id of a base / quote pair
given a specified round.
| getNextRoundId | Returns the next round id of a base / quote pair
given a specified round.
| getCurrentPhaseId | Returns the current phase id of a base / quote pair.
|

```

---

#### decimals

Get the number of decimals present in the response value.

```

{/ prettier-ignore /}
solidity
function decimals(address base, address quote) external view returns (uint8)

```

#### Parameters

- base: The base asset address.
- quote: The quote asset address.

#### Return values

- RETURN: The number of decimals.

#### description

Get the description of the underlying aggregator that the proxy points to.

```

{/ prettier-ignore /}
solidity
function description(address base, address quote) external view returns (string
memory)

```

#### Parameters

- base: The base asset address.
- quote: The quote asset address.

#### Return values

- RETURN: The description of the underlying aggregator.

## getRoundData

Get data about a specific round, using the roundId.

```


{
 // prettier-ignore
}
solidity
function getRoundData(address base, address quote, uint80 roundId) external view
 returns (
 uint80 roundId,
 int256 answer,
 uint256 startedAt,
 uint256 updatedAt,
 uint80 answeredInRound
)

```

### Parameters

- base: The base asset address.
- quote: The quote asset address.
- roundId: The round ID.

### Return values

- roundId: The round ID.
- answer: The price.
- startedAt: Timestamp of when the round started.
- updatedAt: Timestamp of when the round was updated.
- answeredInRound:  Deprecated - Previously used when answers could take multiple rounds to be computed

## latestRoundData


Get the price from the latest round.

```

{
 // prettier-ignore
}
solidity
function latestRoundData(address base, address quote) external view
 returns (
 uint80 roundId,
 int256 answer,
 uint256 startedAt,
 uint256 updatedAt,
 uint80 answeredInRound
)

```

### Return values

- roundId: The round ID.
- answer: The price.
- startedAt: Timestamp of when the round started.
- updatedAt: Timestamp of when the round was updated.
- answeredInRound:  Deprecated - Previously used when answers could take multiple rounds to be computed

## version

The version representing the type of aggregator the proxy points to.

```

{
 // prettier-ignore
}
solidity
function version(address base, address quote) external view returns (uint256)

```



#### Parameters

- base: The base asset address.
- quote: The quote asset address.

#### Return values

- RETURN: The version number.

#### getFeed

Returns the primary aggregator address of a base / quote pair. Note that onchain contracts cannot read from aggregators directly, only through Feed Registry or Proxy contracts.

```
{/ prettier-ignore /}
solidity
function getFeed(address base, address quote) external view returns
(AggregatorV2V3Interface aggregator);
```

#### Parameters

- base: The base asset address.
- quote: The quote asset address.

#### Return values

- aggregator: The primary aggregator address.

#### getPhaseFeed

Returns the underlying aggregator address of a base / quote pair at a specified phase. Note that onchain contracts cannot read from aggregators directly, only through Feed Registry or Proxy contracts.

Phase ids start at 1. You can get the current Phase by calling `getCurrentPhaseId()`.

```
{/ prettier-ignore /}
solidity
function getPhaseFeed(
 address base,
 address quote,
 uint16 phaseId
) external view returns (AggregatorV2V3Interface aggregator);
```

#### Parameters

- base: The base asset address.
- quote: The quote asset address.
- phaseId: The phase id.

#### Return values

- aggregator: The primary aggregator address at the specified phase.

#### isFeedEnabled

Returns true if an aggregator is enabled as primary on the feed registry. This is useful to check if you should index events from an aggregator contract,

because you want to only index events of primary aggregators.

```
{/ prettier-ignore /}
solidity
function isFeedEnabled(address aggregator) external view returns (bool);
```

#### Parameters

- aggregator: The aggregator address

#### Return values

- RETURN: true if the supplied aggregator is a primary aggregator for any base / quote pair.

#### getPhase

Returns the starting and ending aggregator round ids of a base / quote pair.

```
{/ prettier-ignore /}
solidity
function getPhase(address base, address quote, uint16 phaseId) external view
returns (Phase memory phase);
```

Phases hold the following information:

```
{/ prettier-ignore /}
solidity
struct Phase {
 uint16 phaseId;
 uint80 startingAggregatorRoundId;
 uint80 endingAggregatorRoundId;
}
```

#### Parameters

- base: The base asset address.
- quote: The quote asset address.
- phaseId: The phase id.

#### Return values

- RETURN: Phase details of a base / quote pair.

#### getRoundFeed

Returns the underlying aggregator address of a base / quote pair at a specified round. Note that onchain contracts cannot read from aggregators directly, only through Feed Registry or Proxy contracts.

```
{/ prettier-ignore /}
solidity
function getRoundFeed(
 address base,
 address quote,
 uint80 roundId
) external view returns (AggregatorV2V3Interface aggregator);
```

#### Parameters

- base: The base asset address.
- quote: The quote asset address.
- roundId: The round id.

#### Return values

- aggregator: The underlying aggregator address of a base / quote pair at the specified round.

#### getPhaseRange

Returns the starting and ending round ids of a base / quote pair at a specified phase.

Please note that this roundId is calculated from the phase id and the underlying aggregator's round id. To get the raw aggregator round ids of a phase for indexing purposes, please use getPhase().

```
{/ prettier-ignore /}
solidity
function getPhaseRange(
 address base,
 address quote,
 uint16 phaseId
) external view returns (uint80 startingRoundId, uint80 endingRoundId);
```

#### Parameters

- base: The base asset address.
- quote: The quote asset address.
- phaseId: The phase id.

#### Return values

- startingRoundId: The starting round id
- endingRoundId: The ending round id

#### getPreviousRoundId

Returns the previous round id of a base / quote pair given a specified round. Note that rounds are non-monotonic across phases.

```
{/ prettier-ignore /}
solidity
function getPreviousRoundId(address base, address quote, uint80 roundId)
external view returns (uint80 previousRoundId);
```

#### Parameters

- base: The base asset address.
- quote: The quote asset address.
- roundId: The round id.

#### Return values

- previousRoundId: The previous round id of a base / quote pair.

## getNextRoundId

Returns the next round id of a base / quote pair given a specified round. Note that rounds are non-monotonic across phases.

```
{/ prettier-ignore /}
solidity
function getNextRoundId(address base, address quote, uint80 roundId) external
view returns (uint80 nextRoundId);
```

### Parameters

- base: The base asset address.
- quote: The quote asset address.
- roundId: The round id.

### Return values

- nextRoundId: The next round id of a base / quote pair.

## getCurrentPhaseId

Returns the current phase id of a base / quote pair.

```
{/ prettier-ignore /}
solidity
function getCurrentPhaseId(address base, address quote) external view returns
(uint16 currentPhaseId);
```

### Parameters

- base: The base asset address.
- quote: The quote asset address.

### Return values

- phaseId: The current phase id of a base / quote pair.

# index.mdx:

```

section: dataFeeds
date: Last Modified
title: "Feed Registry"
isIndex: true
metadata:
 title: "How to Use Chainlink Feed Registry"
 description: "The Chainlink Feed Registry is an onchain mapping of assets to
feeds. It allows users and DeFi protocols to query Chainlink data feeds from a
given pair of asset and denomination addresses."
whatsnext:
 {
 "Read the Feed Registry API Reference": "/data-feeds/feed-registry/feed-
registry-functions",
 "See the FeedRegistryInterface contract on GitHub":
"https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/
interfaces/FeedRegistryInterface.sol",
 }
}
```

A Denominations Solidity library is available for you to fetch currency identifiers which lack a canonical Ethereum address:

```
<CodeSample src="samples/FeedRegistry/Denominations.sol" />
```

## Code Examples

### Solidity

To consume price data from the Feed Registry, your smart contract should reference `FeedRegistryInterface`, which defines the external functions implemented by the Feed Registry.

```
<CodeSample src="samples/FeedRegistry/PriceConsumer.sol" />
```

### Solidity Hardhat Example

```
<Aside type="note" title="Note">
 You can find a working Feed Registry Hardhat project
 here. Clone the repo and follow the setup instructions to
 run the example locally.
</Aside>
```

### Javascript

```
<CodeSample src="samples/FeedRegistry/PriceConsumer.js" />
```

```
<RegistryPrice
 client:idle
 registryAddress={registryAddresses["link-usd"].network.mainnet.address}
 baseSymbol={registryAddresses["link-usd"].baseSymbol}
 baseAddress={registryAddresses["link-usd"].network.mainnet.baseAddress}
 quoteSymbol={registryAddresses["link-usd"].quoteSymbol}
 quoteAddress={registryAddresses["link-usd"].network.mainnet.quoteAddress}
 supportedChain="ETHEREUMMAINNET"
/>
```

### Contract addresses

This section lists the blockchains that Chainlink Feed Registry are currently available on.

Network	Address
-----	
-----	
Ethereum Mainnet	0x47Fb2585D2C56Fe188D0E6ec628a38b74fCeeDf

```
addresses.mdx:
```

```

title: "Price Feed Contract Addresses"
section: dataFeeds
metadata:
 title: "Price Feed Contract Addresses"
 description: "A list of Price Feed addresses on supported networks."
 date: Last Modified

```

```
import FeedPage from "@features/feeds/components/FeedPage.astro"
```

```
<FeedPage />
```

```
index.mdx:

section: dataFeeds
date: Last Modified
title: "Price Feeds"
isIndex: true
whatsnext:
 {
 "Learn how to read answers from Data Feeds": "/data-feeds/using-data-feeds",
 "Learn how to get Historical Price Data": "/data-feeds/historical-data",
 "Find contract addresses for Price Feeds":
"/data-feeds/price-feeds/addresses",
 "Data Feeds API Reference": "/data-feeds/api-reference",
 }
metadata:
 title: "Using Data Feeds"
 description: "How to use Chainlink Data Feeds in your smart contracts."

```

Chainlink Data Feeds provide data that is aggregated from many data sources by a decentralized set of independent node operators. The Decentralized Data Model describes this in detail. However, there are some exceptions where data for a feed can come only from a single data source or where data values are calculated. Read the Selecting Quality Data Feeds to learn about the different data feed categories and how to identify them.

```
addresses.mdx:

title: "Proof of Reserve Feed Addresses"
section: dataFeeds
datafeedtype: por
metadata:
 title: "Proof of Reserve Feed Addresses"
 description: "Chainlink Proof of Reserve Feed Addresses"
date: Last Modified

import FeedPage from "@features/feeds/components/FeedPage.astro"

<FeedPage dataFeedType="por" />
```

```
index.mdx:

section: dataFeeds
date: Last Modified
title: "Proof of Reserve Feeds"
isIndex: true
whatsnext:
 {
 "Learn how to read answers from Data Feeds": "/data-feeds/price-feeds",
 "Learn how to get Historical Price Data": "/data-feeds/historical-data",
 "Find contract addresses for Proof of Reserve Feeds": "/data-feeds/proof-of-reserve/addresses",
 "Data Feeds API Reference": "/data-feeds/api-reference",
 }

import { Aside, ClickToZoom, CodeSample } from "@components"
```

Chainlink Proof of Reserve Feeds provide the status of the reserves for several assets. You can read these feeds the same way that you read other Data Feeds. Specify the Proof of Reserve Feed Address that you want to read instead of specifying a Price Feed address. See the Using Data Feeds page to learn more.

To find a list of available Proof of Reserve Feeds, see the Proof of Reserve Feed Addresses page.

## Types of Proof of Reserve Feeds

Reserves are available for both cross-chain assets and offchain assets. This categorization describes the data reporting variations of Proof of Reserve feeds and helps highlight some of the inherent market risks surrounding the data quality of these feeds.

Reserves are available for both offchain assets and cross-chain assets. Token issuers prove the reserves for their assets through several different methods.

### Offchain reserves

Offchain reserves are sourced from APIs through an external adapter.

<ClickToZoom src="/images/data-feed/off-chain-reserves.webp" />

Offchain reserves provide their data using the following methods:

- Third-party: An auditor, accounting firm, or other third party audits and verifies reserves. This is done by combining both fiat and investment assets into a numeric value that is reported against the token.
- Custodian: Reserves data are pulled directly from the bank or custodian. The custodian has direct access to the bank or vault holding the assets. Generally, this works when the underlying asset pulled requires no additional valuation and is simply reported onchain.
- Self-reported: Reserve data is read from an API that the token issuer hosts. Reserve data reported by an asset issuer's self-hosted API carries additional risks. Chainlink Labs is not responsible for the accuracy of self-reported reserves data. Users must do their own risk assessment for asset issuer risk.

### Cross-chain reserves

Cross-chain reserves are sourced from the network where the reserves are held. Chainlink node operators can report cross-chain reserves by running an external adapter and querying the source-chain client directly. In some instances, the reserves are composed of a dynamic list of IDs or addresses using a composite adapter.

<ClickToZoom src="/images/data-feed/cross-chain-reserves.webp" />

Cross-chain reserves provide their data using the following methods:

- Wallet address manager: The project uses the IPoRAAddressList wallet address manager contract and self-reports which addresses they own. Reserve data reported by an asset issuer's self-reported addresses carries additional risks. Chainlink Labs is not responsible for the accuracy of self-reported reserves data. Users must do their own risk assessment for asset issuer risk.
- Wallet address: The project reports which addresses they own through a self-hosted API. Reserve data reported by an asset issuer's self-reported addresses carries additional risks. Chainlink Labs is not responsible for the accuracy of self-reported reserves data. Users must do their own risk assessment for asset issuer risk.

## Using Proof of Reserve Feeds



Read answers from Proof of Reserve Feeds the same way that you read other Data Feeds. Specify the Proof of Reserve Feed Address that you want to read instead of specifying a Price Feed address. See the Using Data Feeds page to learn more.

Using Solidity, your smart contract should reference AggregatorV3Interface, which defines the external functions implemented by Data Feeds.

```
<CodeSample src="samples/DataFeeds/ReserveConsumerV3.sol" />
```

```
<Aside title="Disclaimer" type="caution">
```

```
<p>
```

Proof of Reserve feeds can vary in their configurations. Please be careful with the configuration of the feeds used

by your smart contracts. You are solely responsible for reviewing the quality of the data (e.g., a Proof of Reserve

feed) that you integrate into your smart contracts and assume full responsibility for any damage, injury, or any

other loss caused by your use of the feeds used by your smart contracts.

```

```

Learn more about making responsible data quality decisions.

```

```

```
</p>
```

```
</Aside>
```

```
addresses.mdx:
```

```

```

```
title: "Rate and Volatility Feed Addresses"
```

```
section: dataFeeds
```

```
datafeedtype: rates
```

```
metadata:
```

```
 title: "Rate and Volatility Feed Addresses"
```

```
 description: "Address lists for Rate and Volatility feeds."
```

```
date: Last Modified
```

```

```

```
import FeedPage from "@features/feeds/components/FeedPage.astro"
```

```
<FeedPage dataFeedType="rates" />
```

```
index.mdx:
```

```

```

```
section: dataFeeds
```

```
date: Last Modified
```

```
title: "Rate and Volatility Feeds"
```

```
isIndex: true
```

```
whatsnext:
```

```
{
```

```
 "Learn how to read answers from Data Feeds": "/data-feeds/price-feeds",
```

```
 "Learn how to get Historical Price Data": "/data-feeds/historical-data",
```

```
 "Find contract addresses for Rate and Volatility Feeds": "/data-feeds/rates-feeds/addresses",
```

```
 "Data Feeds API Reference": "/data-feeds/api-reference",
```

```
}
```

```

```

```
import { ClickToZoom } from "@components"
```

Chainlink rate and volatility feeds provide data for interest rates, interest rate curves, and asset volatility. You can read these feeds the same way that you read other Data Feeds. Specify the Rate or Volatility Feed Address that you

want to read instead of specifying a Price Feed address. See the Using Data Feeds page to learn more.

The following data types are available:

- Bitcoin Interest Rate Curve
- ETH Staking APR
- Realized Volatility

#### Bitcoin Interest Rate Curve

Lenders and borrowers use base rates to evaluate interest rate risk for lending and borrowing contracts, asset valuation for derivatives contracts, and an underlying rate for interest rate swap contracts. Bitcoin Interest Rate Curve Data Feeds provide a base rate to assist with market decisions and quantify the risks of using certain protocols and products based on current and predicted baseline interest rates. The curve's normalized methodology and daily rates introduce more consistency and predictability to the ebb and flow of digital asset markets. Bitcoin Interest Rate Curve Feeds incorporate a wide range of data sources such as OTC lending desks, DeFi lending pools, and perpetual futures markets.

To learn more about the use of these interest rate curves in the industry, read the Bitcoin Interest Rate Curve (CF BIRC) blog post.

See the Rate and Volatility Feed Addresses page to find the Bitcoin Interest Rate Curve feeds that are currently available.

#### ETH Staking APR

The ETH Staking APR feeds provide a trust-minimized and tamper-proof source of truth for the global rate of return from staking as a validator to secure the Ethereum network. The annualized rate of return is calculated over 30-day and 90-day rolling windows. Data providers use offchain computation to calculate returns at an epoch level, reach consensus on the APR, and then write the results onchain to be used by decentralized protocols and Web 3 applications. Feeds are currently configured to update at a minimum of once per day.

```
<ClickToZoom
 src="/images/data-feed/eth-staking-apr-feed-V3.webp"
 alt="A diagram showing how Staking data is obtained and annualized rate of
return is confirmed onchain"
/>
```

See the Rate and Volatility Feed Addresses page to find the ETH Staking APR feeds that are currently available. If you have questions or would like to request an enhancement to ETH Staking APR feeds, contact us using this form.

#### Realized volatility

Realized volatility measures asset price movement over a specific time interval. This value is expressed as a percent of the asset price. The more an asset price moves up or down over time, the higher the realized volatility is for that asset. Please note that realized volatility is not the same as implied volatility, which measures the market's expectation about future volatility typically derived from options markets.

Each data feed reflects the volatility of an asset over a specific rolling window of time. For example, some data feeds provide volatility data for the last 24 hours, 7 days, and 30 days of time. You can compare the data across these windows to infer whether the volatility of an asset is trending up or down. For example, if realized volatility for the 24-hour window is higher than the 7-day window, volatility might increase.

The same high-quality data providers used in Chainlink<sup>™</sup>s price feeds sample price data every 10 minutes to refresh volatility estimates. onchain values are updated when the feed heartbeat or deviation threshold is met.

```
<ClickToZoom
 src="/images/data-feed/realised-volatility-feed-V3.webp"
 alt="A diagram showing how volatility data is obtained and answers for
different volatility windows is confirmed onchain"
/>
```

See the [Rate and Volatility Feed Addresses](#) page to find heartbeat and deviation information for each feed. If you have questions or would like to request an enhancement to Realized Volatility Feeds, contact us using [this form](#).

```
index.mdx:
```

```

section: dataFeeds
date: Last Modified
title: "Data Feeds on Solana"
isIndex: true
whatsnext:
 {
 "Use data feeds offchain": "/data-feeds/solana/using-data-feeds-off-chain",
 "Use data feeds onchain": "/data-feeds/solana/using-data-feeds-solana",
 "See the available data feeds on Solana":
"/data-feeds/price-feeds/addresses?network=solana",
 }

```

```
import { Aside } from "@components"
```

Chainlink provides data feeds on the Solana network. Chainlink data feeds on Solana employ Offchain Reporting (OCR) to aggregate data from data providers who pull from both centralized and decentralized exchanges. Chainlink<sup>™</sup>s Solana deployment has no dependencies on external blockchain networks such as Ethereum. In Solana, storage and smart contract logic are separate. Programs store all the logic similar to an EVM (Ethereum) smart contract. The accounts store all the data. Compared to Solidity, the combination of an account and a program is equivalent to a smart contract on an EVM chain. State and logic are separate in Solana.

Solana programs are stateless, so you don't always need to deploy your program to the network to test it. You can deploy and test your programs on a Solana Test Validator. However, to use Chainlink products on Solana, you must deploy your contract onchain to one of the supported Solana clusters.

```
<Aside type="note" title="Note">
```

```
 Please note that Price Feeds performance relies on the chains they are
 deployed on. Periods of high network congestion
 may impact the frequency of Chainlink Price Feeds. Subscribe to Solana status
 notifications to stay updated on system performance.
</Aside>
```

To learn how to mitigate risk to your applications, read the [Selecting Quality Data Feeds](#) page.

Chainlink products and Solana clusters

Price Feeds are available on the following Solana clusters:

- Solana Mainnet
- Solana Devnet

Solana provides a Testnet cluster that runs newer Solana releases, but Chainlink Data Feeds are not available on this cluster.

See the Solana Data Feeds page for a full list of Chainlink data feeds that are available on Solana.

To learn when more Chainlink services become available, follow us on Twitter or sign up for our mailing list.

## Languages, tools, and frameworks

The examples in the Chainlink documentation use the following languages, tools, and frameworks:

- Node.js 14 or higher: Used to run client code
- Rust: A general-purpose programming language designed for performance and memory safety
- Anchor: A Solana Sealevel Framework that provides several developer tools
- Chainlink Solana Starter Kit: An Anchor based program and client that shows developers how to use and interact with Chainlink Data Feeds on Solana
- Solana CLI: The Solana command line interface
- Git: Used to clone the example code repository

When developing applications to use Chainlink products on Solana, always use a Mainnet release version of the Solana CLI that is equal to or greater than the version currently running on your target cluster. Use `solana --version` and `solana cluster-version` to check CLI and cluster versions:

```
shell
solana --version
solana-cli 1.9.28 (src:b576e9cc; feat:320703611)
```

```
solana cluster-version --url devnet
1.9.25
```

```
solana cluster-version --url mainnet-beta
1.9.28
```

The examples in this documentation use Solana programs in Rust, but you can also write Solana programs in C. To learn more about the Solana programming model, see the Solana Documentation.

## Solana wallets

When you use Chainlink on Solana, you need a Solana wallet. The Chainlink documentation uses file system wallets and free Devnet SOL tokens to demonstrate examples. When you deploy your programs to the Solana Mainnet, you must use wallets with mainnet lamports.

If you have existing wallets that you want to use for the guides in the Chainlink documentation, find your wallet keypair and make it available in your development environment as a file. You can point Anchor and the Solana CLI to a specific keypair when you deploy or manage your Solana programs.

```
shell
anchor build
â€®
```

```
anchor deploy --provider.wallet ./config/solana/id.json --provider.cluster
devnet
â€®
```

```
solana program show --programs --keypair /.config/solana/id.json --url devnet
```

```
Program Id | Slot | Authority
| Balance
6U4suTp55kiJRKqV7HGAQvFgcLaStLnUA4myg5DRqsKw | 109609728 |
E6gKKToCJPgf4zEL1GRLL6T99g2WcfAzJAMvtma1KijT | 2.57751768 SOL
```

When you build your production applications and deploy Solana programs to the Mainnet cluster, always follow the security best practices in the Solana Wallet Guide for managing your wallets and keypairs.

```
using-data-feeds-off-chain.mdx:
```

```

section: dataFeeds
date: Last Modified
title: "Using Data Feeds Offchain (Solana)"
whatsnext:
 {
 "Use data feeds onchain": "/data-feeds/solana/using-data-feeds-solana",
 "See the available data feeds on Solana":
"/data-feeds/price-feeds/addresses?network=solana",
 }
metadata:
 title: "Using Data Feeds Offchain (Solana)"
 description: "How to use Chainlink Data Feeds in your offchain applications."

```

```
import { Aside, CodeSample, PackageManagerTabs } from "@components"
import { Tabs } from "@components/Tabs"
```

Chainlink Data Feeds are the quickest way to access market prices for real-world assets. This guide demonstrates how to read Chainlink Data Feeds on the Solana Devnet using offchain examples in the Chainlink Solana Starter Kit. To learn how to use Data Feeds in your onchain Solana programs, see the Using Data Feeds onchain guide.

To get the full list of Chainlink Data Feeds on Solana, see the Solana Feeds page.

```
<Aside type="danger" title="Select quality data feeds">
```

Be aware of the quality of the data that you use. Learn more about making responsible data quality decisions.

```
</Aside>
```

### The Chainlink Data Feeds Store Program

The program that contains the logic required for the storing and retrieval of Chainlink Data Feeds data on both Devnet and Mainnet is `cjg3oHmg9uuPsP8D6g29NWvhySJkdYdAo9D25PRbKXJ`. This is the program ID that you use to read price data from offchain. You can find the source code for this program in the `smartcontractkit/chainlink-solana` on GitHub.

You can add data feeds to an existing offchain project or use the Solana Starter Kit.

### Adding Data Feeds to an existing offchain project

You can read Chainlink Data Feeds offchain in your existing project by using the Chainlink Solana NPM library.

<Aside type="caution" title="Reading feed data">

Although you can directly query the data feed accounts, you should not rely on the memory layout always being the same as it currently is. Based on this, the recommendation is to always use the consumer library.

</Aside>

Install the necessary components and include the example code in your project. Optionally, you can run the example code by itself to learn how it works before you integrate it with your project.

1. Install the latest Mainnet version of the Solana CLI and export the path to the CLI:

```
shell
sh -c "$(curl -sSfL https://release.solana.com/v1.13.6/install)" &&
export PATH="/.local/share/solana/install/active-release/bin:$PATH"
```

Run `solana --version` to make sure the Solana CLI is installed correctly.

```
shell
solana --version
```

1. Install Node.js 14 or higher. Run `node --version` to verify which version you have installed:

```
shell
node --version
```

1. Change to your project directory or create a new directory.

```
shell
mkdir off-chain-project && cd off-chain-project
```

1. Optionally install Yarn to use as a package manager and initialize yarn if your project does not already have a package.json file:

```
shell
npm install -g yarn && yarn init
```

1. Add the Anchor library to your project:

```
{/ prettier-ignore /}
<PackageManagerTabs>
<Fragment slot="yarn">
shell yarn
yarn add @project-serum/anchor

</Fragment>
<Fragment slot="npm">
shell npm
npm i @project-serum/anchor

</Fragment>
</PackageManagerTabs>
```

1. Add the Chainlink Solana NPM library to your project:

```

{/ prettier-ignore /}
<PackageManagerTabs>
<Fragment slot="yarn">
shell yarn
yarn add @chainlink/solana-sdk

</Fragment>
<Fragment slot="npm">
shell npm
npm i -g @chainlink/solana-sdk

</Fragment>
</PackageManagerTabs>

```

1. Create a temporary Solana wallet to use for this example. Alternatively, if you have an existing wallet that you want to use, locate the path to your keypair file and use it as the keypair for the rest of this guide.

```

shell
solana-keygen new --outfile ./id.json

```

1. Set the Anchor environment variables. Anchor uses these to determine which wallet to use and how to get a connection to a Solana cluster. Because this example does not generate or sign any transactions, no lamports are required. The wallet is required only by the Anchor library. For a list of available networks and endpoints, see the Solana Cluster RPC Endpoints documentation.

```

shell
export ANCHORPROVIDERURL=https://api.devnet.solana.com &&
export ANCHORWALLET=./id.json

```

1. Copy the sample code into your project. This example queries price data offchain. By default, the script reads the SOL/USD feed, but you can change the CHAINLINKFEEDADDRESS variable to point to the feed account addresses that you want to query. You can take the components of these code samples and integrate them with your existing project. Because these examples read data feeds without making any onchain changes, no lamports are required to run them.

```
<CodeSample src="samples/Solana/PriceFeeds/off-chain-read.js" />
```

```
<CodeSample src="samples/Solana/PriceFeeds/off-chain-read.ts" />
```

You can run these examples using the following commands:

```

{/ prettier-ignore /}
<Tabs client:visible>
<Fragment slot="tab.js">Javascript</Fragment>
<Fragment slot="tab.ts">Typescript</Fragment>
<Fragment slot="panel.js">
shell javascript
node javascript-example.js

</Fragment>
<Fragment slot="panel.ts">
shell typescript
yarn add ts-node typescript && yarn ts-node typescript-example.ts

</Fragment>
</Tabs>

```

To learn more about Solana and Anchor, see the Solana Documentation and the

Anchor Documentation.

## Using the Solana Starter Kit

This example reads price data from an offchain client using the Solana Starter Kit.

Install the required tools

Before you begin, set up your environment for development on Solana:

1. Install Git if it is not already configured on your system.

1. Install the latest Mainnet version of the Solana CLI and export the path to the CLI:

```
shell
sh -c "$(curl -sSfL https://release.solana.com/v1.13.6/install)" &&
export PATH="/.local/share/solana/install/active-release/bin:$PATH"
```

Run `solana --version` to make sure the Solana CLI is installed correctly.

```
shell
solana --version
```

1. Install Node.js 14 or higher. Run `node --version` to verify which version you have installed:

```
shell
node --version
```

1. Install Anchor. On some operating systems, you might need to build and install Anchor locally. See the Anchor documentation for instructions.

1. Install Yarn to simplify package management and run code samples in the Starter Kit.

```
shell
npm install -g yarn
```

Run the example program

After you install the required tools, clone the example code from the `solana-starter-kit` repository.

1. In a terminal, clone the `solana-starter-kit` repository and change to the `solana-starter-kit` directory:

```
shell
git clone https://github.com/smartcontractkit/solana-starter-kit &&
cd ./solana-starter-kit
```

You can see the complete code for the example on GitHub.

1. In the `./solana-starter-kit` directory, install Node.js dependencies defined in the `package.json` file:

```
shell
yarn install
```



1. Create a temporary Solana wallet file to use for this example. Because your application runs offchain and does not run any functions or alter data onchain, the wallet does not require any SOL tokens to function.

```
shell
solana-keygen new --outfile ./id.json
```

1. Set the Anchor environment variables. Anchor uses these to determine which wallet to use and Solana cluster to use. Take note that because we are not generating or signing any transactions, the wallet isn't used, it's just required by the Anchor library. For a list of available networks and endpoints, see the Solana Cluster RPC Endpoints documentation.

```
shell Solana Devnet
export ANCHORPROVIDERURL=https://api.devnet.solana.com &&
export ANCHORWALLET=./id.json
```

1. Run the example:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.js">Javascript</Fragment>
 <Fragment slot="tab.ts">Typescript</Fragment>
 <Fragment slot="panel.js">
 shell JavaScript
 node read-data.js

 </Fragment>
 <Fragment slot="panel.ts">
 shell TypeScript
 yarn run read-data

 </Fragment>
</Tabs>
```

The example code retrieves and prints the current price feed data until you close the application:

```
4027000000
4026439929
4026476542
4023000000
```

To learn more about Solana and Anchor, see the Solana Documentation and the Anchor Documentation.

```
using-data-feeds-solana.mdx:
```

```

section: dataFeeds
date: Last Modified
title: "Using Data Feeds Onchain (Solana)"
whatsnext:
 {
 "Use Data Feeds Offchain": "/data-feeds/solana/using-data-feeds-off-chain",
 "See the available data feeds on Solana":
 "/data-feeds/price-feeds/addresses?network=solana",
```

```

}
metadata:
 title: "Using Data Feeds Onchain (Solana)"
 description: "How to use Chainlink Data Feeds in your onchain Solana
programs."

```

```

import { Aside, CodeSample } from "@components"
import { Tabs } from "@components/Tabs"
```

Chainlink Data Feeds are the quickest way to connect your smart contracts to the real-world market prices of assets. This guide demonstrates how to deploy a program to the Solana Devnet cluster and access Data Feeds onchain using the Chainlink Solana Starter Kit. To learn how to read price feed data using offchain applications, see the Using Data Feeds Offchain guide.

To get the full list of available Chainlink Data Feeds on Solana, see the Solana Feeds page. View the program that owns the Chainlink Data Feeds in the Solana Devnet Explorer, or the Solana Mainnet Explorer.

```
<Aside type="danger" title="Select quality data feeds">
```

Be aware of the quality of the data that you use. Learn more about making responsible data quality decisions.

```
</Aside>
```

### The Chainlink Data Feeds OCR2 program

The program that owns the data feeds on both Devnet and Mainnet is HEvSKofvBgfaev23kMabbYqxasxU3mQ4ibBMEJWHny. This is the program ID that you use to retrieve Chainlink Price Data onchain in your program. The source code for this program is available in the smartcontractkit/chainlink-solana repository on GitHub.

You can add data feeds to an existing project or use the Solana Starter Kit.

### Adding Data Feeds onchain in an existing project

You can read Chainlink Data Feed data onchain in your existing project using the Chainlink Solana Crate.

```
<Aside type="caution" title="Reading feed data">
```

Although you can directly query the data feed accounts, you should not rely on the memory layout always being the same as it currently is. Based on this, the recommendation is to always use the consumer library queries below.

```
</Aside>
```

Import the Chainlink Solana Crate into your project and use the code sample to make function calls.

1. Add the Chainlink Solana Crate as an entry in your Cargo.toml file dependencies section, as shown in the starter kit Cargo.toml example.

```

toml
[dependencies]
chainlinksolana = "1.0.0"
```

1. Use the following code sample to query price data. Each function call to the Chainlink Solana library takes two parameters:

- The feed account that you want to query.
- The Chainlink Data Feeds OCR2 Program for the network. This is a static value that never changes.

The code sample has the following components:

- latestrounddata: Returns the latest round information for the specified price pair including the latest price
- description: Returns a price pair description such as SOL/USD
- decimals: Returns the precision of the price, as in how many numbers the price is padded out to
- Display: A helper function that formats the padded out price data into a human-readable price

```
<CodeSample src="samples/Solana/PriceFeeds/on-chain-read.rs" />
```

```
<CodeSample src="samples/Solana/PriceFeeds/on-chain-read-anchor.rs" />
```

Program Transaction logs:

```
{/ prettier-ignore /}
<Tabs client:visible>
 <Fragment slot="tab.rust">Rust</Fragment>
 <Fragment slot="tab.rustAnchor">Rust with Anchor</Fragment>
 <Fragment slot="panel.rust">
 shell Rust
 > Program logged: "Chainlink Price Feed Consumer entrypoint" > Program
logged: "SOL / USD price is
83.99000000"
 > Program consumed: 95953 of 1400000 compute units > Program return:
HNYsBr77Jc9LhHeb9tx53SrWbWfNBnQzQrM4b3BB3PCR CA==

 </Fragment>
 <Fragment slot="panel.rustAnchor">
 shell Rust with Anchor
 Fetching transaction logs... ['Program
HEvSKofvBgfaexv23kMabbYqxasxU3mQ4ibBMEJWHny
consumed 1826 of 1306895 compute units', 'Program return:
HEvSKofvBgfaexv23kMabbYqxasxU3mQ4ibBMEJWHny CA==',
'Program HEvSKofvBgfaexv23kMabbYqxasxU3mQ4ibBMEJWHny success', 'Program
log: SOL / USD price is 93.76988029',]

 </Fragment>
</Tabs>
```

To learn more about Solana and Anchor, see the [Solana Documentation](#) and the [Anchor Documentation](#).

Using the Solana starter kit

This guide demonstrates the following tasks:

- Write and deploy programs to the Solana Devnet cluster using Anchor.
- Retrieve price data using the Solana Web3 JavaScript API with Node.js.

This example shows a full end to end example of using Chainlink Price Feeds on Solana. It includes an onchain program written in rust, as well as an offchain client written in JavaScript. The client passes in an account to the program, the program then looks up the latest price of the specified price feed account, and then stores the result in the passed in account. The offchain client then reads the value stored in the account.

Install the required tools

Before you begin, set up your environment for development on Solana:

1. Install Git if it is not already configured on your system.

1. Install Node.js 14 or higher. Run `node --version` to verify which version you have installed:

```
shell
node --version
```

1. Install Yarn to simplify package management and run code samples.

1. Install a C compiler such as the one included in GCC. Some of the dependencies require a C compiler.

1. Install Rust:

```
shell
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh &&
source $HOME/.cargo/env
```

1. Install the latest Mainnet version of the Solana CLI and export the path to the CLI:

```
shell
sh -c "$(curl -sSfL https://release.solana.com/v1.13.6/install)" &&
export PATH="/.local/share/solana/install/activerelease/bin:$PATH"
```

Run `solana --version` to make sure the Solana CLI is installed correctly.

```
shell
solana --version
```

1. Install Anchor. On some operating systems, you might need to build and install Anchor locally. See the Anchor documentation for instructions.

After you install the required tools, build and deploy the example program from the `solana-starter-kit` repository.

Deploy the example program

This example includes a contract written in Rust. Deploy the contract to the Solana Devnet cluster.

1. In a terminal, clone the `solana-starter-kit` repository and change to the `solana-starter-kit` directory:

```
shell
git clone https://github.com/smartcontractkit/solana-starter-kit &&
cd ./solana-starter-kit
```

You can see the complete code for the example on GitHub.

1. In the `./solana-starter-kit` directory, install Node.js dependencies defined in the `package.json` file:

```
shell
yarn install
```

1. Create a temporary Solana wallet to use for this example. Use a temporary wallet to isolate development from your other wallets and prevent you from unintentionally using lamports on the Solana Mainnet. Alternatively, if you have an existing wallet that you want to use, locate the path to your keypair file and use it as the keypair for the rest of this guide.

```
shell
solana-keygen new --outfile ./id.json
```

When you build your production applications and deploy Solana programs to the Mainnet cluster, always follow the security best practices in the Solana Wallet Guide for managing your wallets and keypairs.

1. Fund your Solana wallet. On Devnet, use solana airdrop to add tokens to your account. The contract requires at least 4 SOL to deploy and the faucet limits each request to 2 SOL, so you must make two requests to get a total of 4 SOL on your wallet:

```
shell
solana airdrop 2 --keypair ./id.json --url devnet &&
solana airdrop 2 --keypair ./id.json --url devnet
```

- If the command line faucet does not work, run solana address on the temporary wallet to print the public key value for the wallet and request tokens from SolFaucet:

```
shell
solana address -k ./id.json
```

1. Run anchor build to build the example program. If you receive the no such subcommand: 'build-bpf' error, restart your terminal session and run anchor build again:

```
shell
anchor build
```

1. The build process generates the keypair for your program's account. Before you deploy your program, you must add this public key to the lib.rs file:

1. Get the keypair from the ./target/deploy/chainlinksolanademo-keypair.json file that Anchor generated:

```
shell
solana address -k ./target/deploy/chainlinksolanademo-keypair.json
```

1. Edit the ./programs/chainlinksolanademo/src/lib.rs file and replace the keypair in the declareid!() definition:

```
shell
vi ./programs/chainlinksolanademo/src/lib.rs

shell
declareid!("JC16qi56dgcLoaTve4BvnCoDL6FhH5NtahA7jmWZFdqM");
```

1. With the new program ID added, run anchor build again. This recreates the necessary program files with the correct program ID:

```
shell
anchor build
```

1. Run anchor deploy to deploy the program to the Solana Devnet. Remember to specify the keypair file for your wallet and override the default. This wallet is the account owner (authority) for the program:

```
shell
anchor deploy --provider.wallet ./id.json --provider.cluster devnet
```

1. To confirm that the program deployed correctly, run solana program show --programs to get a list of deployed programs that your wallet owns. For this example, check the list of deployed programs for the id.json wallet on the Solana Devnet:

```
shell
solana program show --programs --keypair ./id.json --url devnet
```

The command prints the program ID, slot number, the wallet address that owns the program, and the program balance:

```
shell
Program Id | Slot | Authority
| Balance
GRT21UnJFHZvcaWLbcUrXaTCFMREewDrm1DweDYBak3Z | 110801571 |
FsQPnANKDhqpoayxCL3oDHFCBmrhP34NrfbDR34qbQUt | 3.07874904 SOL
```

To see additional details of your deployed program, copy the program ID and look it up in the Solana Devnet Explorer.

Now that the program is onchain, you can call it.

Call the deployed program

Use your deployed program to retrieve price data from a Chainlink data feed on Solana Devnet. For this example, call your deployed program using the client.js example code.

1. Set the Anchor environment variables. Anchor uses these to determine which wallet to use and Solana cluster to use.

```
shell
export ANCHORPROVIDERURL=https://api.devnet.solana.com &&
export ANCHORWALLET=./id.json
```

1. Run the client.js example and pass the program address in using the --program flag:

```
shell
node client.js --program $(solana address -k
./target/deploy/chainlinksolanademo-keypair.json)
```

If the script executes correctly, you will see output with the current price of SOL / USD.

```
shell
â€®
```

```
Price Is: 96.79778375
Success
â.®
```

1. Each request costs an amount of SOL that is subtracted from the id.json wallet. Run solana balance to check the remaining balance for your temporary wallet on Devnet.

```
shell
solana balance --keypair ./id.json --url devnet
```

1. To get prices for a different asset pair, run client.js again and add the --feed flag with one of the available Chainlink data feeds. For example, to get the price of BTC / USD on Devnet, use the following command:

```
shell
node client.js \
--program $(solana address -k ./target/deploy/chainlinksolanademo-
keypair.json) \
--feed CZZQBrJCLqjXRfMjRN3fhbxur2QYHUzkpaRwkWsiPqbz
```

```
shell
Price Is: 12.4215826
Success
```

The program that owns the data feeds is HEvSKofvBgfaexv23kMabbYqxasxU3mQ4ibBMEJWHny, which you can see defined for const CHAINLINKPROGRAMID in the client.js file.

## Clean up

After you are done with your deployed contract and no longer need it, it is nice to close the program and withdraw the Devnet SOL tokens for future use. In a production environment, you will want to withdraw unused SOL tokens from any Solana program that you no longer plan to use, so it is good to practice the process when you are done with programs on Devnet.

1. Run solana program show to see the list of deployed programs that your wallet owns and the balances for each of those programs:

```
shell
solana program show --programs --keypair ./id.json --url devnet
```

```
shell
Program Id | Slot | Authority
| Balance
GRt21UnJFHZvcaWLbcUrXaTCFMREewDrm1DweDYBak3Z | 110801571 |
FsQPnANKDhqpoyxCL3oDHFCBmrhP34NrfbDR34qbQut | 3.07874904 SOL
```

1. Run solana program close and specify the program that you want to close:

```
shell
solana program close [YOURPROGRAMID] --keypair ./id.json --url devnet
```

The program closes and the remaining SOL is transferred to your temporary wallet.

1. If you have deployments that failed, they might still be in the buffer holding SOL tokens. Run solana program show again with the --buffers flag:

```
shell
solana program show --buffers --keypair ./id.json --url devnet
```

If you have open buffers, they will appear in the list.

```
shell
Buffer Address | Authority
| Balance
CSc9hnBqYJoYtBgsryJAmrjAE6vZ918qaFhL6N6BdEmB |
FsQPnANKDhqpoyxCL3oDHFCBmrhP34NrfbDR34qbQUt | 1.28936088 SOL
```

1. If you have any buffers that you do not plan to finish deploying, run the same solana program close command to close them and retrieve the unused SOL tokens:

```
shell
solana program close [YOURPROGRAMID] --keypair ./id.json --url devnet
```

1. Check the balance on your temporary wallet.

```
shell
solana balance --keypair ./id.json --url devnet
```

1. If you are done using this wallet for examples and testing, you can use solana transfer to send the remaining SOL tokens to your default wallet or another Solana wallet that you use. For example, if your default wallet keypair is at /.config/solana/id.json, you can send ALL of the temporary wallet's balance with the following command:

```
shell
solana transfer /.config/solana/id.json ALL --keypair ./id.json --url devnet
```

Alternatively, you can send the remaining balance to a web wallet. Specify the public key for your wallet instead of the path the default wallet keypair. Now you can use those Devnet funds for other examples and development.

To learn more about Solana and Anchor, see the Solana Documentation and the Anchor Documentation.

# index.mdx:

```

section: dataFeeds
date: Last Modified
title: "Using Data Feeds on Starknet"
isIndex: true
whatsnext:
 {
 "Starknet Price Feed Contract Addresses":
"/data-feeds/price-feeds/addresses?network=starknet",
 "Deploy and interact with a Consumer Contract using Starknet Foundry":
"/data-feeds/starknet/tutorials/snfoundry/consumer-contract",
 }
metadata:
 description: "Explore how to use Chainlink Data Feeds on Starknet to power
```



your applications with data."

---

```
import { CopyText } from "@components"
```

Starknet is a permissionless decentralized Zk-rollup operating as an L2 network over Ethereum. Unlike other Ethereum L2 networks, Starknet is not EVM-compatible and uses Cairo as its smart contract language. Chainlink Data Feeds are available on Starknet Sepolia as Cairo smart contracts.

#### Offchain examples

You can read Chainlink Data Feeds offchain without a Starknet account. Complete these steps using only the Starkli CLI or the Starknet Foundry toolkit.

#### Onchain examples

You can read Chainlink Data Feeds on Starknet Sepolia using an onchain contract that you compile, declare, and deploy using Starknet Foundry.

#### Devnet examples

Experiment with Chainlink Data Feeds on Starknet using Starknet Devnet RS, a local Docker-based testnet environment implemented using Rust. The Starknet Devnet RS guide provides code examples and scripts that enable you to compile, declare, and deploy your own aggregator and consumer contracts. You can use a set of pre-funded accounts to interact with Chainlink Data Feeds on Starknet without deploying to a live network.

#### Getting Started (Starkli CLI)

Starkli is a Starknet standalone CLI similar to cairo-lang, but it is written in Rust. It enables interaction with a Starknet network without the need for extra tools or dependencies.

In this example, you will use the starkli call command to read data from a Chainlink Price Feed on Starknet Sepolia. This command requires the proxy aggregator address, the function selector, and an RPC endpoint for Starknet Sepolia.

#### Requirements

Make sure you have the Starkli CLI installed. You can check your current version by running `<CopyText text="starkli --version" code/>` in your terminal. Expect an output similar to the following:

```
bash
starkli --version
0.2.8 (f59724e)
```

Follow the official installation guide>) if necessary.

#### Read data from a Chainlink Price Feed

Use the following command to call the latestrounddata function on the ETH / USD Chainlink Price Feed aggregator proxy:

```
bash
starkli call \
0x228128e84cdfc51003505dd5733729e57f7d1f7e54da679474e73db4ecaad44 \
latestrounddata \
--rpc https://starknet-sepolia.public.blastapi.io/rpc/v07
```

[illegible][illegible]

Note: This example uses a Blast API RPC endpoint. You can interact with the network using any other Starknet Sepolia RPC provider, such as Alchemy or Infura.

```
consumer-contract.mdx:

section: dataFeeds
date: Last Modified
title: "Deploy and Interact with a Consumer Contract using Starknet Foundry"
whatsnext:
 {
 "Starknet Price Feed Contract Addresses":
"/data-feeds/price-feeds/addresses?network=starknet",
 "Experiment with Starknet Devnet RS":
"/data-feeds/starknet/tutorials/snfoundry/sn-devnet-rs",
 }
metadata:
```

description: "Deploy and Interact with a Data Feeds Consumer Contract on StarkNet using Starknet Foundry."  
---

```
import { CodeSample, CopyText } from "@components"
```

This example uses a local OpenZeppelin account to deploy and interact with a consumer contract onchain. This contract retrieves data from a specified Chainlink data feed on Starknet Sepolia and stores the information for the latest round of data.

## Requirements

### Set up your environment

This guide uses the Starknet Foundry toolkit and the Scarb project management tool so you can compile, deploy, and interact with your Starknet smart contracts.

- Starknet Foundry: Install Starknet Foundry v0.21.0. You can check your current version by running `<CopyText text="snforge --version" code/>` or `<CopyText text="sncast --version" code/>` in your terminal and install the required version if necessary.

- Scarb: Install Scarb v2.6.4. You can check your current version by running `<CopyText text="scarb --version" code/>` in your terminal and install the required version if necessary.

Alternatively, you can use the asdf tool version manager to install both Starknet Foundry and Scarb. Read the setup instructions for Starknet Foundry and Scarb for more information.

### Clone and configure the code examples repository

1. Clone the chainlink-starknet repository, which includes the example contracts for this guide:

```
bash
git clone https://github.com/smartcontractkit/chainlink-starknet.git
```

1. Navigate to the aggregatorconsumer directory:

```
bash
cd chainlink-starknet/examples/contracts/aggregatorconsumer/
```

After you prepare the requirements, make sure the required tools are configured correctly by running the tests:

```
bash
make test
```

The contracts should compile successfully and the tests should pass.

## Tutorial

### Create, deploy, and fund an account

1. Run the create-account script to create a new OpenZeppelin local account for the Sepolia testnet:

```
bash
```

```
make create-account
```

The account details are stored locally in your `/.starknetaccounts/starknetopenzeppelinaccounts.json` file.

Expect an output similar to the following:

```
bash
command: account create
address: 0x3e7ee3d05d3efae4e493c9733c8c391d4a5cc63d34f95a164c832060fcdd43d
maxfee: 9529927566560
message: Account successfully created. Prefund generated address with at
least <maxfee> tokens. It is good to send more in the case of higher demand.
```

Your accounts:

```
{
 "alpha-sepolia": {
 "testnet-account": {
 "address":
"0x3e7ee3d05d3efae4e493c9733c8c391d4a5cc63d34f95a164c832060fcdd43d",
 "classhash":
"0x4c6d6cf894f8bc96bb9c525e6853e5483177841f7388f74a46cfda6f028c755",
 "deployed": false,
 "legacy": false,
 "privatekey": <PRIVATEKEY>,
 "publickey":
"0x2972c3cb7aa85403fa1e038f9ce7a93f025ea57017eb7ef735bae989bfb15bd",
 "salt": "0x61a062d2dd4e7656"
 }
 }
}
```

From the output, note:

- Your account address. In this example, it is `0x03e7ee3d05d3efae4e493c9733c8c391d4a5cc63d34f95a164c832060fcdd43d`, with an added zero after `0x`.
- The `maxfee` value, which is the minimum amount of testnet ETH required to deploy the account.

1. Fund the newly created account with testnet ETH to cover the account deployment, the consumer contract deployment, and the network interaction costs.

Go to the Blast Starknet Sepolia ETH Faucet and enter your account address to receive testnet ETH. Wait a few seconds for the transaction to complete.

Alternatively, you can transfer ETH tokens from a Starknet-compatible wallet, such as Argent or Braavos, or use the StarkGate bridge to transfer testnet ETH from Ethereum Sepolia to Starknet Sepolia. If you need testnet ETH on Ethereum Sepolia, you can use the Chainlink faucet.

1. Deploy your account to Starknet Sepolia:

```
bash
make deploy-account
```

Expect an output similar to the following:

```
bash
command: account deploy
transactionhash:
```

0x541ffae49f77dd942376391286052d9fb87dc8d6848139ab1508c114e3aa005

Wait a few seconds for the transaction to complete.

Deploy and interact with a consumer contract

The consumer contract contains two functions that you interact with in this example:

- setanswer retrieves the latest answer from the specified aggregator contract and stores it in the answer internal storage variable.
- readanswer reads the stored answer value.

1. Navigate to /scripts/src/consumer/ and open the deployaggregatorconsumer script.

1. Update the aggregatoraddress variable within the main function with the ETH / USD Chainlink proxy aggregator contract address on Starknet Sepolia : <CopyText text="0x228128e84cdfc51003505dd5733729e57f7d1f7e54da679474e73db4ecaad44" code/>.

1. Run the deployaggregatorconsumer script to deploy a consumer contract to Starknet Sepolia. Use the following command:

```
bash
make ac-deploy NETWORK=testnet
```

Expect an output similar to the following:

```
bash
Declaring and deploying AggregatorConsumer
Declaring contract...

Class hash =
2231728956022539490111802723283413449895905447951536434260474920103409356221

Deploying contract...
Transaction hash =
0x46b39d79cf90ed75fbae5d097d6026cf7ab1184e852d36aed336528fc77220d
Waiting for transaction to be accepted (59 retries / 295s left until timeout)

AggregatorConsumer deployed at address:
2956260449156927152048242588422796467290585049226993045191257886158889626959

command: script run
status: success
```

The consumer address is represented in its decimal form:  
2956260449156927152048242588422796467290585049226993045191257886158889626959.  
You can use a converter tool to get the hexadecimal format. In this example, the hexadecimal equivalent is  
0x06892f22691473dc5a413a7626062604155516ec91a82d3734dc68bcf3d72d4f. Save your consumer address for the next steps.

1. In /scripts/src/consumer/, open the setanswer script and update the consumeraddress variable with your deployed consumer address.

1. Run the following command to update the answer:

```
bash
make ac-set-answer NETWORK=testnet
```

This command runs the `setanswer` script to retrieve the latest answer from the ETH / USD aggregator contract and stores it in the internal storage variable `answer` of your consumer contract.

Expect an output similar to the following:

```
bash
Result::Ok(CallResult { data:
[3374557091160001179079409876363576078229871685244120671563970051729036896526] }
)
Result::Ok(CallResult { data: [340282366920938463463374607431768224268,
354661371569, 54349, 1711716556, 1711716514] })
Transaction hash =
0x20edd6388041d78a36f153a7694fd8c0849d81aff3fc2d5a0606d7275ea4837
Waiting for transaction to be accepted (59 retries / 295s left until timeout)
Result::Ok(InvokeResult { transactionhash:
930889525374955449815155593419397099236202282693723934594292211164864202807 })
command: script run
status: success
```

1. In `/scripts/src/consumer/`, open the `readanswer` script and update the `consumeraddress` variable with your deployed consumer address.

1. Run the following command to read the stored answer value from your consumer contract:

```
bash
make ac-read-answer NETWORK=testnet
```

Expect an output similar to the following:

```
bash
Result::Ok(CallResult { data: [354661371569] })
command: script run
status: success
```

The answer on the ETH / USD feed uses 8 decimal places, so an answer of 354661371569 indicates an ETH / USD price of 3546.61371569. Each feed uses a different number of decimal places for answers. You can find the correct number of decimal places on the Price Feed addresses page by clicking the Show more details checkbox.

# `index.mdx`:

```

section: dataFeeds
date: Last Modified
title: "Data Feeds guides (Starknet Foundry)"
isIndex: true
whatsnext: { "Starknet Price Feed Contract Addresses": "/data-feeds/price-
feeds/addresses?network=starknet" }
metadata:
 description: "Learn how to interact with Chainlink Data Feeds on StarkNet using
Starknet Foundry."

```

You can explore several comprehensive guides to learn how to use Data Feeds on Starknet using Starknet Foundry. These tutorials provide step-by-step guidance to help you understand how to locally experiment with or integrate Chainlink

## Guides

- ```
# read-data.mdx:
```

— — —

[illegible]

The example output contains an array with the hex-encoded latest round of data for the ETH / USD price feed. The array contains the following values:

| Value name | Hex-encoded value | Decoded value |
|-------------|--------------------------------------|---|
| Description | | |
| ----- | ----- | |
| ----- | | ----- |
| roundid | 0x100000000000000000000000000000320c | 340282366920938463463374607431768224268 The unique identifier of the data round |
| answer | 0x5293770eb1 | 354661371569 The actual data provided by the data feed, representing the latest price of an asset in the case of a price feed |
| blocknum | 0xd44d | 54349 The block number at which the data was recorded on the blockchain |
| startedat | 0x6606b8cc | 1711716556 The Unix timestamp indicating when the data round started |
| updatedat | 0x6606b8a2 | 1711716514 The Unix timestamp indicating when the data was last updated |

For a complete list of Chainlink Price Feeds available on Starknet, see the [Price Feed Contract Addresses](#) page.

Note: This example uses a Blast API RPC endpoint. You can interact with the network using any other Starknet Sepolia RPC provider, such as Alchemy or Infura.

```
# sn-devnet-rs.mdx:
```

```

---
section: dataFeeds
date: Last Modified
title: "Experiment with Data Feeds using Starknet Foundry and Starknet Devnet
RS"
whatsnext: { "Starknet Price Feed Contract Addresses": "/data-feeds/price-
feeds/addresses?network=starknet" }
metadata:
  description: "Experiment with Chainlink Data Feeds on a Devnet using Starknet
Devnet RS."
---

```

```
import { CodeSample, CopyText, Aside } from "@components"
```

This guide employs Starknet Devnet RS, a local Docker-based testnet environment for Starknet. It provides code examples, scripts, and commands that enable you to deploy and experiment with your own aggregator and consumer contracts. You can use a set of prefunded accounts to interact with Chainlink Data Feeds on Starknet without deploying to a live network.

Requirements

Set up your environment

This guide uses the Starknet Foundry toolkit and the Scarb project management tool so you can compile, deploy, and interact with your Starknet smart contracts.

- Starknet Foundry: Install Starknet Foundry v0.21.0. You can check your current version by running `<CopyText text="snforge --version" code/>` or `<CopyText text="sncast --version" code/>` in your terminal and install the required version if necessary.

- Scarb: Install Scarb v2.6.4. You can check your current version by running `<CopyText text="scarb --version" code/>` in your terminal and install the required version if necessary.

Alternatively, you can use the asdf tool version manager to install both Starknet Foundry and Scarb. Read the setup instructions for Starknet Foundry and Scarb for more information.

- Docker: Install Docker Desktop. You will run Starknet Devnet RS in a Docker container.

Clone and configure the code examples repository

1. Clone the chainlink-starknet repository, which includes the example contracts for this guide:

```
bash
git clone https://github.com/smartcontractkit/chainlink-starknet.git
```

1. Navigate to the aggregatorconsumer directory:

```
bash
cd chainlink-starknet/examples/contracts/aggregatorconsumer/
```

1. Open your snfoundry.toml file and locate the [sncast.devnet] section. Make sure the url points to the correct endpoint for connecting to the Starknet node running inside the Docker container. By default, it is:

```
bash
[sncast.default]
url = "http://127.0.0.1:5050/rpc"
```

After you prepare the requirements, check to make sure the required tools are configured correctly by running the tests:

```
bash
make test
```

The contracts should compile successfully and the tests should pass.

Tutorial

<Aside type="note" title="Contract addresses for Starknet Devnet RS">

If you followed the onchain guide, clone the chainlink-starknet repository again to make sure you start from a clean state. By default, some addresses used in this guide are hardcoded to work with the Starknet Devnet RS environment.

</Aside>

Run Starknet Devnet RS and add a prefunded account

1. Make sure Docker Desktop is running.

1. Execute the following command to run a Starknet Devnet RS container:

```
bash
make devnet
```

Note: Executing this command when a container is running stops and recreates the container, which helps restart from a clean state.

1. The Starknet devnet container includes a set of prefunded accounts. To execute the scripts from this guide, add one of these accounts to a local accounts.json file by running the following command:

```
bash
make add-account
```

Expect an output similar to the following:

```
bash
Importing a prefunded account from starknet devnet container...
```

```
command: account add
```

Your accounts:

```
{
  "alpha-goerli": {
    "devnet-account": {
      "address":
"0x4b3f4ba8c00a02b66142a4b1dd41a4dfab4f92650922a3280977b0f03c75ee1",
      "classhash":
"0x61dac032f228abef9c6626f995015233097ae253a7f72d68552db02f2971b8f",
      "deployed": true,
      "legacy": false,
      "privatekey": "0x57b2f8431c772e647712ae93cc616638",
      "publickey":
"0x374f7fcb50bc2d6b8b7a267f919232e3ac68354ce3eafe88d3df323fc1deb23"
    }
  }
}
```

Deploy and interact with a mock aggregator contract

1. Declare and deploy a mock aggregator contract to the devnet by using the deploymockaggregator script. Run the following command:

```
bash
make ma-deploy NETWORK=devnet
```

Expect an output similar to the following:

```
bash
Declaring and deploying MockAggregator
Declaring contract...
Transaction hash =
0x1d4cb925322e0f9645cf0bf4028b7fca963b5325cc685784c0f6e22628de8ad
Class hash =
3238358970964595466962588940073124312643094641461979698784985925711454737896
Deploying contract...
Transaction hash =
0x6b065766a26105fbd9bf55be46675d2c2b4b0dd9a85db5ab0b6a1f8ad04f743
```

MockAggregator deployed at address:
1708487128334545444076626254389547159540422020354796381590079145465639129383

```
command: script run
status: success
```

1. Read the latest round of data by using the readlatestround script. Run the following command:

```
bash
make agg-read-latest-round NETWORK=devnet
```

Expect an output similar to the following:

```
bash
Result::Ok(CallResult { data: [0, 0, 0, 0, 0] })
command: script run
status: success
```

The default data on the mock aggregator contract is all set to zeros.

1. Set the latest round of data to values defined in the setlatestround script. Run the following command:

```
bash
make ma-set-latest-round NETWORK=devnet
```

Expect an output similar to the following:

```
bash
Transaction hash =
0x3f01f4595dcf20b4f355720540d5279efc0fff306d4e848dda4550cdd2a5367
Result::Ok(InvokeResult { transactionhash:
1781197671452105104356220458359085564429259248315865550090042149835146679143 })
command: script run
status: success
```

1. Read the latest round of data again to verify the updated values:

```
bash
make agg-read-latest-round NETWORK=devnet
```

Expect an output similar to the following:

```
bash
Result::Ok(CallResult { data: [1, 1, 12345, 1711716556, 1711716514] })
command: script run
status: success
```

Deploy and interact with a mock aggregator and a mock consumer contract

The mock aggregator consumer contract is initialized with the address of an aggregator mock contract during the contract's constructor call and contains two functions:

- `setanswer` retrieves the latest answer from the specified aggregator contract and stores it in the internal storage variable `answer`.

- readanswer reads the stored answer value.

1. Run the following command to (re)start the Starknet Devnet RS container:

```
bash
make devnet
```

1. If you haven't already, add a prefunded account to the local accounts.json file using the following command:

```
bash
make add-account
```

1. Declare and deploy both a mock aggregator and a consumer contracts to the devnet by running the following command:

```
bash
make devnet-deploy
```

This command executes both deploymockaggregator and deployaggregatorconsumer scripts.

Expect an output similar to the following:

```
bash
Declaring and deploying MockAggregator
Declaring contract...
Transaction hash =
0x568d29d07128cba750845b57a4bb77a31f628b6f4288861d8b31d12e71e4c3b
Deploying contract...
Transaction hash =
0xfbc49eb82894a704ce536ab904cdee0fd021b0fba335900f8b9b12cfcd005f
MockAggregator deployed at address:
1566652744716179301065270359129119857774335542042051464747302084192731701184

command: script run
status: success

Declaring and deploying AggregatorConsumer
Declaring contract...
Transaction hash =
0x3f11f08103e263690b1eeac76c25ce7b003c86d2d1c0492815d1ddb39f58e8e
Deploying contract...
Transaction hash =
0xfe5b1ed51a435117098343b5d0fdc1c32fd493f1220ff2c69c732f5bb805d8
AggregatorConsumer deployed at address:
2775662320989421053891107164335108610292525354452508848326323790006553228656

command: script run
status: success
```

Once the contracts are deployed, experiment with both contracts and the available scripts and commands as needed for your projects.

For instance, you can set new values for the latest round of data on the aggregator contract, retrieve and set the answer on the consumer contract, and read the answer from the consumer contract:

```
bash
make ma-set-latest-round NETWORK=devnet && make ac-set-answer NETWORK=devnet &&
```

```
make ac-read-answer NETWORK=devnet
```

```
# architecture.mdx:
```

```
---
section: dataStreams
title: "Data Streams Architecture"
whatsnext:
  {
    "Find the list of available stream IDs.": "/data-streams/stream-ids",
    "Find the schema of data to expect from Data Streams reports.": "/data-
streams/reference/report-schema",
  }
---
```

```
import { Aside, ClickToZoom } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<DataStreams section="dsNotes" />
```

High Level Architecture

Chainlink Data Streams has the following core components:

- A Chainlink Decentralized Oracle Network (DON): This DON operates similarly to the DONs that power Chainlink Data Feeds, but the key difference is that it signs and delivers reports to the Chainlink Data Streams Aggregation Network rather than delivering answers onchain directly. This allows the Data Streams DON to deliver reports more frequently for time-sensitive applications. Nodes in the DON retrieve data from many different data providers, reach a consensus about the median price of an asset, sign a report including that data, and deliver the report to the Data Streams Aggregation Network.
- The Chainlink Data Streams Aggregation Network: The Data Streams Aggregation Network stores the signed reports and makes them available for retrieval. The network uses an active-active multi-site deployment to ensure high availability and robust fault tolerance by operating multiple active sites in parallel. The network delivers these reports to Chainlink Automation upon request (Streams Trade) or provide direct access via the API (Streams Direct).
- The Chainlink Verifier Contract: This contract verifies the signature from the DON to cryptographically guarantee that the report has not been altered from the time that the DON reached consensus to the point where you use the data in your application.

Streams Trade Architecture

Using Chainlink Automation with Data Streams automates trade execution and mitigates frontrunning by executing the transaction before the data is recorded onchain. Chainlink Automation requests data from the Data Streams Aggregation Network. It executes transactions only in response to the data and the verified report, so the transaction is executed correctly and independently from the decentralized application itself.

```
<ClickToZoom
  src="/images/data-streams/data-streams-trade-architecture-v3.webp"
  alt="Chainlink Data Streams - Streams Trade Architecture"
/>
```

Example trading flow using Streams Trade

One example of how to use Data Streams with Automation is in a decentralized exchange. An example flow might work using the following process:

```
<ClickToZoom
  src="/images/data-streams/streams-trade-sequence-diagram.webp"
  alt="Chainlink Data Streams - Streams Trade Example Trading Flow"
/>
```

1. A user initiates a trade by confirming an initiateTrade transaction in their wallet.
1. The onchain contract for the decentralized exchange responds by emitting a log trigger event.
1. The Automation upkeep monitors the contract for the event. When Automation detects the event, it runs the checkLog function specified in the upkeep contract. The upkeep is defined by the decentralized exchange.
1. The checkLog function uses a revert with a custom error called StreamsLookup. This approach aligns with EIP-3668 and conveys the required information through the data in the revert custom error.
1. Automation monitors the StreamsLookup custom error that triggers Data Streams to process the offchain data request. Data Streams then returns the requested signed report in the checkCallback function for Automation.
1. Automation passes the report to the Automation Registry, which executes the performUpkeep function defined by the decentralized exchange. The report is included as a variable in the performUpkeep function.
1. The performUpkeep function calls the verify function on the Data Streams onchain verifier contract and passes the report as a variable.
1. The verifier contract returns a verifierResponse bytes value to the upkeep.
1. If the response indicates that the report is valid, the upkeep executes the user's requested trade. If the response is invalid, the upkeep rejects the trade and notifies the user.

This is one example of how you can combine Data Streams and Automation, but the systems are highly configurable. You can write your own log triggers or custom logic triggers to initiate Automation upkeeps for a various array of events. You can configure the StreamsLookup to retrieve multiple reports. You can configure the performUpkeep function to perform a wide variety of actions using the report.

Read the Getting Started guide to learn how to build your own smart contract that retrieves reports from Data Streams using the Streams Trade implementation.

Streams Direct Architecture

```
<DataStreams section="streamsDirectEarlyAccess" />
```

Example of offchain price updates with Streams Direct

Streams Direct enables seamless offchain price updates through a mechanism designed for real-time data delivery. Here is an example of how your Client will benefit from low-latency market data directly from the Data Streams Aggregation Network.

1. The Client opens a WebSocket connection to the Data Streams Aggregation Network to subscribe to new reports with low latency.
1. The Data Streams Aggregation Network streams price reports via WebSocket, which gives the Client instant access to updated market data.
1. The Client stores the price reports in a cache for quick access and use, which preserves data integrity over time.
1. The Client regularly queries the Data Streams Aggregation Network for any missed reports to ensure data completeness.
1. The Aggregation Network sends back an array of reports to the Client.
1. The Client updates its cache to backfill any missing reports, ensuring the

data set remains complete and current.

```
<ClickToZoom  
  src="/images/data-streams/streams-direct-offchain-price-updates.webp"  
  alt="Chainlink Data Streams - Streams Direct Off-Chain Price Updates"  
>
```

Active-Active Multi-Site Deployment

Active-active is a system configuration strategy where redundant systems remain active simultaneously to serve requests. Incoming requests are distributed across all active resources and load-balanced to provide high availability, scalability, and fault tolerance. This strategy is the opposite of active-passive where a secondary system remains inactive until the primary system fails.

The Data Streams API services use an active-active setup as a highly available and resilient architecture across multiple distributed and fully isolated origins. This setup ensures that the services are operational even if one origin fails, which provides robust fault tolerance and high availability. This configuration applies to both the REST API and the WebSocket API. A global load balancer seamlessly manages the system to provide automated and transparent failovers. For advanced use cases, the service publishes available origins using HTTP headers, which enables you to interact directly with specific origin locations if necessary.

Active-Active Setup

The API services are deployed across multiple distributed data centers. Each active deployment is fully isolated and capable of handling requests independently. This redundancy ensures that the service can withstand the failure of any single site without interrupting service availability.

Global Load Balancer

A global load balancer sits in front of the distributed deployments. The load balancer directs incoming traffic to the healthiest available site based on real-time health checks and observed load.

- Automated Failover: In the event of a site failure, traffic is seamlessly rerouted to operational sites without user intervention.
- Load Distribution: Requests are balanced across all active sites to optimize resource usage and response times.

Origin Publishing

To enable advanced interactions, the service includes the origin information for all of the available origins in the HTTP headers of API responses. This feature allows customers to explicitly target specific deployments if desired. It also allows for concurrent WebSocket consumption from multiple sites, ensuring fault tolerant WebSocket subscriptions, low-latency, and minimized risk of report gaps.

Example Failover Scenarios

Automatic failover handles availability and traffic routing in the following scenarios:

- Automatic Failover: If one of the origins becomes unavailable, the global load balancer automatically reroutes traffic to the next available origin. This process is transparent to the user and ensures uninterrupted service. During automatic failover, WebSockets experience a reconnect. Failed REST requests must be retried.

- Manual Traffic Steering: If you want to bypass the load balancer and target a specific site, you can use the origin headers to direct your requests. This manual targeting does not affect the automated failover capabilities provided by the load balancer, so a request will succeed even if the specified origin is unavailable.

- Multi-origin concurrent WebSocket subscriptions: In order to maintain a highly available and fault tolerant report stream, you can subscribe to up to two available origins simultaneously. This compares the latest consumed timestamp for each feed and discards duplicate reports before merging the report stream locally.

billing.mdx:

```
---
section: dataStreams
title: "Data Streams Billing"
isIndex: false
whatsnext:
  {
    "Learn the basics about how to retrieve Data Streams reports in the Getting Started guide.": "/data-streams/getting-started",
    "Find the list of available stream IDs.": "/data-streams/stream-ids",
    "Find the schema of data to expect from Data Streams reports.": "/data-streams/reference/report-schema",
  }
---
```

import { Aside, ClickToZoom } from "@components"

```
<Aside type="note" title="Talk to an expert">
  <a href="https://chainlinkcommunity.typeform.com/datastreams?#refid=docs">Contact us</a> to talk to an expert about
    integrating Chainlink Data Streams with your applications.
</Aside>
```

You pay to verify reports from Data Streams onchain using the verifier contract. You pay per report verified. If you verify multiple reports in a batch, you pay for all of the reports included in that batch.

The verification price is 0.35 USD per report. Chainlink Data Streams supports fee payments in LINK and in alternative assets, which currently includes native blockchain gas tokens and their ERC20-wrapped version. Payments made in alternative assets have a 10% surcharge when compared to LINK payments.

Contact us to learn more about Mainnet pricing.

developer-responsibilities.mdx:

```
---
section: dataStreams
title: "Developer Responsibilities: Market Integrity and Application Code Risks"
whatsnext:
  {
    "Find the list of available Data Streams Feed IDs": "/data-streams/stream-ids",
    "Find the schema of data to expect from Data Streams reports": "/data-streams/reference/report-schema",
    "Learn the basics about how to retrieve Data Streams reports using the Streams Trade implementation": "/data-streams/getting-started",
    "Learn how to fetch and decode Data Streams reports using the Streams Direct API": "/data-streams/tutorials/streams-direct/streams-direct-api",
  }
---
```



```
}  
---
```

```
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<DataStreams section="dsNotes" />
```

Chainlink Data Streams provide access to high-frequency market data backed by decentralized, fault-tolerant, and transparent infrastructure, where offchain data can be pulled onchain and verified by Chainlink's Verify Contract, as needed by your application. The assets priced by Chainlink Data Streams are subject to market conditions beyond the ability of Chainlink node operators to control. As such, developers are responsible for understanding market conditions and other external risks and how they can impact their products and services.

When integrating Chainlink Data Streams, developers must understand that the performance of individual Streams is subject to risks associated with both market integrity and application code.

- Market Integrity Risks are those associated with external market conditions impacting price behavior and data quality in unanticipated ways. Developers are solely responsible for monitoring and mitigating any potential market integrity risks.
- Application Code Risks are those associated with the quality, reliability, and dependencies of the code on which an application operates. Developers are solely responsible for monitoring and mitigating any potential application code risks related to their own products and services.

See the Market Manipulation vs. Oracle Exploits article for information about market integrity risks and how developers can protect their applications.

Developer Responsibilities

Developers are responsible for maintaining the security and user experience of their applications. They must also securely manage all interactions between their applications and third-party services.

In particular, developers implementing Chainlink Data Streams in their code and applications are responsible for their application's market integrity and code risks that may cause unanticipated pricing data behavior. These are described below in more detail:

Market Integrity Risks

Market conditions can impact the pricing behavior of assets in ways beyond the ability of Chainlink node operators to predict or control.

Market integrity risk factors can include, but are not limited to, market manipulation such as Spoofing, Ramping, Bear Raids, Cross-Market Manipulation, Washtrading, and Frontrunning. Developers are solely responsible for accounting for such risk factors when integrating Chainlink Data Streams into their applications. Developers should understand the market risks around the assets they intend their application to support before integrating associated Chainlink Data Streams and inform their end users about applicable market risks.

Traditional Market Assets

Among other assets, Data Streams supports foreign exchange (FX) spot markets for major currencies against USD, gold and silver spot against USD, and WTI oil spot contracts. In traditional finance, these markets are some of the most liquid instruments and are traded across most financial centers including London, New York, Tokyo, and Singapore. Unlike Crypto markets, most traditional markets do not trade 24/7 and therefore liquidity and spreads can vary during the day and trading week. For assets traded within traditional markets, Chainlink Data

Streams provides developers with an indication of the market's status (either open or closed), such as via a market status flag in price reports, which can be used by applications as applicable.

The market status provided on Streams serves as an indication of the open and close hours for traditional market assets based on historical practice; it is provided for referential purposes only. Developers are responsible for independently assessing the risks associated with trading at these times, particularly at opening and closing price levels. Developers are solely responsible for determining the actual status of markets for any data feeds they utilize. Protocol developers are advised to proceed with caution and make trading decisions at their own risk.

Under the shared responsibility model, it is essential that developers understand the methodology, trading behavior and risks for the traditional market asset classes and instruments they are supporting and offering to their end users. Data Streams developers are solely responsible for defining and implementing their own risk procedures and systems, including being aware of market open and closing times, and bank holidays, when integrating associated Chainlink Data Streams.

DEX-based Assets

Data Streams also provides pricing data related to assets that trade, primarily, on decentralized exchanges (DEXs). Under the Shared Responsibility model, it is essential that developers understand the methodology and risks associated with such DEX-based assets. The risks include, but are not limited to:

- Data may be sourced by reading the state of onchain contracts and estimating the price at which trades in a certain asset pool could be executed. The accuracy of prices may be hindered by such factors as:
 - Slippage: The movement of prices between (i) the time of observation, and (ii) the time of execution, caused by other transactions changing the state of the respective smart contract.
 - Price Impact: The movement of price caused by the volume of trades being settled on a respective pool.
- Certain assets may not trade activelyâif data is based on traded prices, it may not reflect the current state of the respective pool, and/or the current realizable price.
- There is a certain level of latency between (i) the observability of price data on DEXs, and (ii) the price data being reflected in our price feeds. This can increase the risk of frontrunning.

Application Code Risks

Developers implementing Chainlink Data Streams are solely responsible for instituting risk mitigations, including, but not limited to, data quality checks, circuit breakers, and appropriate contingency logic for their use case. Some general guidelines include:

- Code quality and reliability: Developers must execute code using Chainlink Data Streams only if their code meets the quality and reliability requirements for their use case and application.
- Code and application audits: Developers are responsible for auditing their code and applications before deploying to production. Developers must determine the quality of any audits and ensure that they meet the requirements for their application.
- Code dependencies and imports: Developers are responsible for ensuring the quality, reliability, and security of any dependencies or imported packages that they use with Chainlink Data Streams, and review and audit these dependencies and packages.
- Contingency Logic: In extreme circumstances, including situations outside the control of Chainlink node operators, Chainlink Data Streams may experience periods of unavailability or performance degradation. Developers are responsible

for implementing contingency plans for such circumstances specific to their application, such as the use of the active-active SDK for Data Streams, a secondary fallback oracle, and/or circuit breakers to stall trading.

```
# getting-started-hardhat.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Getting Started with Chainlink Data Streams (Hardhat CLI)"
metadata:
  linkToWallet: true
excerpt: "Learn the basics for how to get data from Chainlink Data Streams."
whatsnext: {
  "Find the list of available stream IDs.": "/data-streams/stream-ids",
  "Find the schema of data to expect from Data Streams reports.":
    "/data-streams/reference/report-schema",
  "Learn more about Log Trigger upkeep.": "/chainlink-automation/guides/log-
trigger/",
}
---
```

```
import { Aside, PageTabs } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<PageTabs
  pages={[
    {
      name: "Remix",
      url: "/data-streams/getting-started",
      icon: "/images/tutorial-icons/remix-icn.png",
    },
    {
      name: "Hardhat",
      url: "/data-streams/getting-started-hardhat",
      icon: "/images/tutorial-icons/hardhat-icn.png",
    },
  ]}
/>
```

```
<DataStreams section="dsNotes" />
```

```
<DataStreams section="gettingStartedHardhat" />
```

```
# getting-started.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Getting Started with Chainlink Data Streams (Remix)"
metadata:
  linkToWallet: true
excerpt: "Learn the basics for how to get data from Chainlink Data Streams."
whatsnext: {
  "Find the list of available Data Streams Feed IDs": "/data-streams/stream-ids",
  "Find the schema of data to expect from Data Streams reports":
    "/data-streams/reference/report-schema",
  "Learn more about Log Trigger upkeep.": "/chainlink-automation/guides/log-
trigger/",
}
---
```

```
import { Aside, PageTabs } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<PageTabs
  pages={[
    {
      name: "Remix",
      url: "/data-streams/getting-started",
      icon: "/images/tutorial-icons/remix-icn.png",
    },
    {
      name: "Hardhat",
      url: "/data-streams/getting-started-hardhat",
      icon: "/images/tutorial-icons/hardhat-icn.png",
    },
  ]}
/>
```

```
<DataStream section="dsNotes" />
```

```
<DataStream section="gettingStarted" />
```

```
# index.mdx:
```

```
---
section: dataStreams
title: "Chainlink Data Streams"
isIndex: true
whatsnext:
  {
    "Learn the basics about how to retrieve Data Streams reports using the
Streams Trade implementation": "/data-streams/getting-started",
    "Learn how to fetch and decode Data Streams reports using the Streams Direct
API": "/data-streams/tutorials/streams-direct/streams-direct-api",
    "Find the list of available Data Streams Feed IDs": "/data-streams/stream-
ids",
    "Find the schema of data to expect from Data Streams reports": "/data-
streams/reference/report-schema",
  }
---
```

```
import { Aside, ClickToZoom } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<DataStream section="dsNotes" />
```

Chainlink Data Streams provides low-latency delivery of market data offchain that you can verify onchain. With Chainlink Data Streams, decentralized applications (dApps) now have on-demand access to high-frequency market data backed by decentralized, fault-tolerant, and transparent infrastructure.

Traditional push-based oracles provide regular updates onchain when certain price thresholds or update time periods have been met. Chainlink Data Streams is built using a new pull-based oracle design that maintains trust-minimization using onchain verification.

Comparison to push-based oracles

Chainlink's push-based oracles provide regular updates onchain. Chainlink Data Streams operates as a pull-based oracle where you can retrieve the data in a report and use it onchain any time. Verifying the report onchain confirms that the data was agreed upon and signed by the DON. While many applications benefit from push-based oracles and require data only after it has been verified

onchain, some applications require access to data that is updated at a higher frequency and delivered with lower latency. Pull-based oracles deliver these benefits while still cryptographically signing the data to ensure its veracity.

```
<ClickToZoom
  src="/images/data-streams/push-based-vs-pull-based-oracles.webp"
  alt="Chainlink Data Streams - Push-Based vs Pull-Based Oracles"
/>
```

Additionally, pull-based oracles deliver data onchain more efficiently by retrieving and verifying the data only when the application needs it. For example, a decentralized exchange might retrieve a Data Streams report and verify the data onchain when a user executes a trade. A push-based oracle repeatedly delivers data onchain even when that data is not immediately required by users.

Comprehensive market insights

Chainlink Data Streams provides price points such as mid prices and Liquidity-Weighted Bid and Ask (LWBA) prices. LWBA prices adjust dynamically based on the current state of order books, which offers greater insights into market liquidity and depth. Liquidity-weighted prices enhance trading accuracy, improve risk management, and increase transactional efficiency as they dynamically reflect the true market conditions based on volume and liquidity.

High availability and resilient infrastructure

The Data Streams API services use an active-active multi-site deployment setup as a highly available and resilient architecture across multiple distributed and fully-isolated origins. This setup ensures that the services are operational even if one origin fails, which provides robust fault tolerance and high availability.

Use cases

Pull-based oracles allow decentralized applications to access data that is updated at a high frequency and delivered with low latency, which enables several new use cases:

- Perpetual Futures: Low-latency data and frontrunning prevention enable onchain perpetual futures protocols that can compete on performance with centralized exchanges while still using more transparent and decentralized infrastructure.
- Options: Pull-based oracles allow timely and precise settlement of options contracts. Additionally, Data Streams provides more detailed market liquidity data that can support dynamic onchain risk management logic.
- Prediction Markets: Higher frequency data updates allow for applications where users can act quickly in response to real-time events and be confident in the accuracy of the data used in the settlement.

Data Streams implementations

Streams Trade: Using Data Streams with Chainlink Automation

When combined with Chainlink Automation, Chainlink Data Streams allows decentralized applications to automate trade execution, mitigate frontrunning, and limit bias or adverse incentives in executing non-user-triggered orders.

```
<ClickToZoom
  src="/images/data-streams/data-streams-trade-architecture-v3.webp"
  alt="Chainlink Data Streams - Streams Trade Architecture"
/>
```

Read more about the Streams Trade Architecture and an example trading flow, or learn how to get started with Streams Trade.

Streams Direct: Using Data Streams with your own bot

Streams Direct offers a direct approach to integrating low-latency and high-frequency data into your applications. You can use the Data Streams SDK to fetch reports (REST API) or to subscribe to report updates (WebSocket connection) from the Data Streams Aggregation Network, and an onchain smart contract to verify reports.

For instance, you can use Chainlink Data Streams with the Streams Direct implementation to display indicative pricing offchain on your front end or with your bot to settle trades onchain.

On-demand offchain workflows

```
<ClickToZoom
  src="/images/data-streams/data-streams-on-demand-offchain-workflows.webp"
  alt="Chainlink Data Streams - Streams Direct On-Demand Offchain Workflows"
/>
```

Explore an example of offchain price updates through Streams Direct in the Architecture guide, or follow this guide to learn how to fetch and decode Data Streams reports using the Data Streams SDK.

```
# release-notes.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Release Notes"
whatsnext:
  {
    "Learn the basics about how to retrieve Data Streams reports in the Getting
    Started guide.": "/data-streams/getting-started",
    "Find the list of available stream IDs.": "/data-streams/stream-ids",
    "Find the schema of data to expect from Data Streams reports.": "/data-
    streams/reference/report-schema",
  }
---
```

```
import { Aside } from "@components"
```

```
<Aside type="note" title="Talk to an expert">
  <a href="https://chainlinkcommunity.typeform.com/datastreams?
#refid=docs">Contact us</a> to talk to an expert about
  integrating Chainlink Data Streams with your applications.
</Aside>
```

2024-08-15 - Base

Chainlink Data Streams is available in Early Access on Base Mainnet and Base Sepolia testnet. Verifier proxy addresses and Feed IDs are available on the [Data Streams Feed IDs](/data-streams/stream-ids) page.

2024-08-02 - Data Streams SDK for Go

The Data Streams SDK for Go is now available.

2024-06-27 - Avalanche

Chainlink Data Streams is available in Early Access on Avalanche Mainnet and Avalanche Fuji testnet. Verifier proxy addresses and Feed IDs are available on the [Data Streams Feed IDs](/data-streams/stream-ids) page.

2024-01-25 - Arbitrum Sepolia

Chainlink Data Streams is available in Early Access on Arbitrum Sepolia testnet. Arbitrum Goerli is no longer supported. Verifier proxy addresses and Feed IDs are available on the [Data Streams Feed IDs](/data-streams/stream-ids) page.

2023-10-02 - Data Streams Early Access

This is the Early Access release for Data Streams on Arbitrum Mainnet and Arbitrum Goerli testnet. Verifier proxy addresses and Feed IDs are available on the [Data Streams Feed IDs](/data-streams/stream-ids) page.

stream-ids.mdx:

```
---
section: dataStreams
title: "Data Streams Feed IDs"
datafeedtype: streams
metadata:
  title: "Data Streams Feed IDs"
  description: "A list of available Data Streams feeds and their ID."
  date: Last Modified
---
```

import FeedPage from "@features/feeds/components/FeedPage.astro"

<FeedPage dataFeedType="streams" initialNetwork="arbitrum" />

liquidity-weighted-prices.mdx:

```
---
section: dataStreams
title: "Data Streams Liquidity-Weighted Bid-Ask Prices (LWBA)"
whatsnext:
  {
    "Find the list of available stream IDs.": "/data-streams/stream-ids",
    "Find the schema of data to expect from Data Streams reports.": "/data-streams/reference/report-schema",
  }
---
```

import DataStreams from "@features/data-streams/common/DataStreams.astro"

<DataStreams section="dsNotes" />

Chainlink Data Streams provides reports with a Mid and Liquidity-Weighted Bid and Ask (LWBA) prices. These three prices form a pricing spread that offers protocols insight into market activity based on the current state of the order books.

Bid and Ask prices

What are Bid and Ask prices?

Bid and Ask prices appear in the order book of an exchange and represent the trading orders buyers and sellers actively submit. These prices drive potential market transactions.

On an exchange, the Mid price in the order book is derived from the midpoint between the best Bid and the best Ask price. While the Mid price serves as a

useful reference, for instance, in calculating funding rates, it is important to note that it is not a price at which trades should be executed. Actual trades must match an open bid or ask.

- Bid price: The Bid price refers to the maximum price that a buyer is willing to pay for an asset. It represents the highest price at which a buyer would agree to buy.

- Ask price: The Ask price is the minimum price at which a seller is willing to sell their asset. It represents the lowest price at which a seller would agree to sell.

Key characteristics

- Price spread: The difference between the Ask and the Bid prices is the spread. A narrower spread typically indicates a more liquid market, whereas a wider spread can signify less liquidity or higher volatility.

- Market orders: When traders place market orders to buy or sell immediately, they accept the best available Ask or Bid prices from the market. This interaction drives immediate transactions but can also impact the market price if the order size is substantial compared to the available volume.

- Liquidity and volume: Bid and Ask prices are not static and can fluctuate based on the asset's trading volume and market liquidity. High liquidity generally results in a narrower spread, which enables more trading activity near this spread and facilitates smoother transactions without substantial impacts on the market price.

Role in Chainlink Data Streams

In Chainlink Data Streams, Bid and Ask prices play a pivotal role in the construction of Liquidity-Weighted Bid and Ask (LWBA) prices.

What is a Liquidity-Weighted price?

A Liquidity-Weighted price considers the Bid and Ask prices based on the available liquidity at each price level in the order books. This method weights price data by the volume of assets available at each price point, and provides a more accurate reflection of market conditions where larger orders would significantly impact the price.

What are the benefits of LWBA prices?

Liquidity-weighted Bid and Ask prices help in several key areas:

- Accuracy: They offer a more precise measure of the market price that considers the volume available at different price levels and allows for true market sentiment to be reflected.

- Risk Management: They consider the market's depth and help protocols assess and manage their risk more effectively and dynamically during periods of high volatility.

- Efficiency: Liquidity-weighted prices facilitate smarter order execution, which accurately reflects realistic slippage and enhances the trading experience for traders and protocols by closely mirroring actual market conditions.

Use case examples

- Derivatives trading: Platforms can use LWBA prices to set more accurate strike prices in options markets or fair settlement prices in futures contracts.

- Liquidity provision: Automated market makers (AMMs) and other liquidity providers can use these prices to quote more accurate and realistic prices that reflect current market conditions.

- Loan collateralization: Lending platforms might use liquidity-weighted prices to determine more realistic valuations for collateral during loan issuance and liquidation processes, thereby reducing the risk of unfair liquidations.

Practical examples of LWBA prices

To better understand how LWBA prices are calculated and used, consider the following examples that illustrate their application in different market scenarios. Each example explains the calculation, highlights trading and risk management benefits, and discusses the spread between Bid and Ask prices.

Example 1: Market volatility

Scenario: Assume the market experiences a sudden drop, leading to a scenario where most buy orders accumulate at a lower price point. Similarly, sellers might be inclined to lower their ask prices to exit positions quickly. The order book shows 10 units at \$0.99 but a more substantial 100 units at \$0.90.

Order book:

Type	Price	Quantity
Bid	\$0.99	10
Bid	\$0.90	100
Ask	\$1.01	15
Ask	\$1.05	85

LWBA Price Calculation:

- Bid = $((10 \times 0.99) + (100 \times 0.90)) / 110 = \0.909
- Ask = $((15 \times 1.01) + (85 \times 1.05)) / 100 = \1.044

Spread Calculation: Ask - Bid = $\$1.044 - \$0.909 = \$0.135$

The LWBA Bid price of \$0.909 and Ask price of \$1.044 provide a realistic snapshot of the market under volatile conditions. The spread of \$0.135 illustrates a wider gap due to high volatility, impacts liquidity and indicates a less efficient market. This dual perspective helps traders and protocols manage risk by pricing assets closer to the most liquid market levels and offers a more cautious approach to valuation during market drops.

Example 2: Market stability

Scenario: In a stable market, traders place the bulk of buy and sell orders close to the current market prices, leading to high liquidity near these levels.

Order book:

Type	Price	Quantity
Bid	\$0.98	90
Bid	\$0.99	10
Ask	\$1.00	50
Ask	\$1.01	50

LWBA Price Calculation:

- Bid = $((10 \times 0.99) + (90 \times 0.98)) / 100 = \0.981
- Ask = $((50 \times 1.00) + (50 \times 1.01)) / 100 = \1.005

Spread Calculation: Ask - Bid = $\$1.005 - \$0.981 = \$0.024$

The LWBA Bid price of \$0.981 and Ask price of \$1.005 reflect the concentration of liquidity near the top price points, with a narrow spread of \$0.024 that indicates a highly liquid and efficient market. This pricing accuracy allows traders to execute orders close to their desired price points with minimal slippage and enhances transaction cost efficiency and market predictiveness in stable conditions.

Conclusion

Example	Weighted Bid Price	Weighted Ask Price	Spread	Market condition
Key benefit				
-----	-----	-----	-----	-----
1	\$0.909	\$1.044	\$0.135	High volatility
Aligns price with majority liquidity				
2	\$0.981	\$1.005	\$0.024	Market stability
Minimizes slippage and improves price accuracy				

In these examples, Liquidity-Weighted Bid and Ask prices, along with the spread between them, adjust dynamically based on the actual distribution of orders within the market. By reflecting real-time liquidity and volume, these prices and spreads provide protocols and traders with critical information that enhances pricing accuracy, reduces trading risks, and increases efficiency across various market conditions.

```
# interfaces.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Interfaces"
---

import { Aside, CodeSample } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"

<DataStreams section="dsNotes" />
```

Data Streams require several interfaces in order to retrieve and verify reports.

- Automation interfaces:
 - StreamsLookupCompatibleInterface
 - ILogAutomation
- Data Streams interfaces:
 - IVerifierProxy
 - IReportHandler

In the current code example for using Data Streams with Automation, these interfaces are specified in the example itself. Imports for these interfaces will be available in the future.

```
<CodeSample src="samples/DataStreams/StreamsUpkeep.sol" />
```

```
# report-schema.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Report Schema"
---

import DataStreams from "@features/data-streams/common/DataStreams.astro"

<DataStreams section="dsNotes" />
```

Data Streams feeds have the following values specifically applicable to market pricing data. Current Data Streams feeds adhere to the schema outlined below:

Value	Type	Description
-----	-----	-----
feedID	bytes32	The unique identifier for the Data Streams feed
validFromTimestamp	uint32	The earliest timestamp during which the price is valid
observationsTimestamp	uint32	The latest timestamp during which the price is valid
nativeFee	uint192	The cost to verify this report onchain when paying with the blockchain's native token
linkFee	uint192	The cost to verify this report onchain when paying with LINK
expiresAt	uint32	The expiration date of this report
price	int192	The DON's consensus median price for this report carried to 18 decimal places
bid	int192	The simulated price impact of a buy order up to the X% depth of liquidity usage
ask	int192	Simulated price impact of a sell order up to the X% depth of liquidity usage

Note: Future Data Streams feeds may use different report schemas.

```
# streams-trade-interface.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Streams Trade Interface"
---
```

```
import { Aside, CodeSample } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<DataStreams section="dsNotes" />
```

To retrieve and verify reports, Streams Trade requires several interfaces.

Automation interfaces

- StreamsLookupCompatibleInterface
- ILogAutomation

Data Streams interfaces

- IVerifierProxy
- IFeeManager

In the code example for using Data Streams with Automation (Streams Trade), the interfaces are specified in the example itself.

```
<CodeSample src="samples/DataStreams/StreamsUpkeep.sol" />
```

```
# index.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Streams Direct Reference"
```

```
isIndex: true
```

```
---
```

```
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<DataStreams section="dsNotes" />
```

- REST API
- WebSocket
- Onchain report data verification

```
# streams-direct-go-sdk.mdx:
```

```
---
```

```
section: dataStreams
```

```
date: Last Modified
```

```
title: "Streams Direct SDK (Go)"
```

```
whatsnext:
```

```
{
  "Learn how to fetch and decode Data Streams reports using the Data Streams
SDK": "/data-streams/tutorials/streams-direct/streams-direct-api",
  "Learn how to stream and decode reports via a WebSocket connection using the
Data Streams SDK": "/data-streams/tutorials/streams-direct/streams-direct-api",
}
```

```
---
```

```
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<DataStreams section="dsNotes" />
```

This documentation provides a detailed reference for the Data Streams SDK for Go. It implements a client library that offers a domain-oriented abstraction for interacting with the Data Streams API and enables both point-in-time data retrieval and real-time data streaming with built-in fault tolerance capabilities.

streams

Installation

```
go
```

```
import streams "github.com/smartcontractkit/data-streams-sdk/go"
```

Types

Client interface

Interface for interacting with the Data Streams API. Use the New function to create a new instance of the client.

Interface Methods

- GetFeeds: Lists all feeds available to you.

```
go
```

```
GetFeeds(ctx context.Context) (r []Feed, err error)
```

- GetLatestReport: Fetches the latest report available for the specified FeedID.

```
go
```

```
GetLatestReport(ctx context.Context, id FeedID) (r Report, err error)
```

- GetReports: Fetches reports for the specified feed IDs and a given timestamp.

```
go
GetReports(ctx context.Context, ids []FeedID, timestamp uint64) ([]Report,
error)
```

- GetReportPage: Paginates the reports for a specified FeedID starting from a given timestamp.

```
go
GetReportPage(ctx context.Context, id FeedID, startTS uint64) (ReportPage,
error)
```

- Stream: Creates a real-time report stream for specified feed IDs.

```
go
Stream(ctx context.Context, feedIDs []FeedID) (Stream, error)
```

Config struct

Configuration struct for the client. Config specifies the client configuration and dependencies. If you specify the Logger function, informational client activity is logged.

```
go
type Config struct {
    ApiKey          string // Client API key
    ApiSecret       string // Client API secret
    RestURL         string // Rest API URL

    WsURL           string // Websocket API URL

    WSHA            bool // Use concurrent connections to multiple Streams
servers
    WsMaxReconnect  int // Maximum number of reconnection attempts for
Stream underlying connections
    LogDebug        bool // Log debug information
    InsecureSkipVerify bool // Skip server certificate chain and host name
verification
    Logger          func(format string, a ...any) // Logger function

    // InspectHttp intercepts http responses for rest requests.
    // The response object must not be modified.
    InspectHttpResponse func(http.Response)
}
```

CtxKey string

Custom context key type used for passing additional headers.

```
go
type CtxKey string
```

- Constants:

- CustomHeadersCtxKey: Key for passing custom HTTP headers.

```

go
const (
    // CustomHeadersCtxKey is used as key in the context.Context object
    // to pass in a custom http headers in a http.Header to be used by the
client.
    // Custom header values will overwrite client headers if they have the same
key.
    CustomHeadersCtxKey CtxKey = "CustomHeaders"
)

```

ReportPage struct

Represents a paginated response of reports.

```

go
type ReportPage struct {
    Reports    []Report // Slice of Report, representing individual report
entries.
    NextPageTS uint64 // Timestamp for the next page, used for fetching
subsequent pages.
}

```

ReportResponse struct

Implements the report envelope that contains the full report payload, its FeedID and timestamps. Use the Decode function to decode the report payload.

```

go
type ReportResponse struct {
    FeedID          feed.ID json:"feedID"
    FullReport      []byte  json:"fullReport"
    ValidFromTimestamp uint64  json:"validFromTimestamp"
    ObservationsTimestamp uint64  json:"observationsTimestamp"
}

```

Methods

- MarshalJSON: Serializes the ReportResponse into JSON.

```

go
func (r ReportResponse) MarshalJSON() ([]byte, error)

```

- String: Returns the string representation of the ReportResponse.

```

go
func (r ReportResponse) String() (s string)

```

- UnmarshalJSON: Deserializes the ReportResponse from JSON.

```

go
func (r ReportResponse) UnmarshalJSON(b []byte) (err error)

```

Stats struct

Statistics related to the Stream's operational metrics.

```

go
type Stats struct {
    Accepted          uint64 // Total number of accepted reports

```

```

    Deduplicated          uint64 // Total number of deduplicated reports when
in HA
    TotalReceived         uint64 // Total number of received reports
    PartialReconnects    uint64 // Total number of partial reconnects when in
HA
    FullReconnects       uint64 // Total number of full reconnects
    ConfiguredConnections uint64 // Number of configured connections if in HA
    ActiveConnections    uint64 // Current number of active connections
}

```

Methods

- String: Returns a string representation of the Stats.

```

go
func (s Stats) String() (st string)

```

Stream interface

Interface for managing a real-time data stream.

Interface Methods

- Read: Reads the next available report from the stream. This method will pause (block) the execution until one of the following occurs:

- A report is successfully received.
- The provided context is canceled, typically due to a timeout or a cancel signal.
- An error state is encountered in any of the underlying connections, such as a network failure.

```

go
Read(context.Context) (Report, error)

```

- Stats: Returns statistics about the stream operations as Stats.

```

go
Stats() Stats

```

- Close: Closes the stream.

```

go
Close() error

```

Functions

New

Creates a new client instance with the specified configuration.

```

go
func New(cfg Config) (c Client, err error)

```

Note: New does not initialize any connections to the Data Streams service.

LogPrintf

Utility function for logging.

```
go
func LogPrintf(format string, a ...any)
```

feed

Installation

```
go
import feed "github.com/smartcontractkit/data-streams-sdk/go/feed"
```

Types

Feed struct

Identifies the report stream ID.

```
go
type Feed struct {
    FeedID ID json:"feedID"
}
```

Where ID is the unique identifier for the feed.

FeedVersion uint16

Represents the feed report schema version.

```
go
type FeedVersion uint16
```

ID [32]byte

Represents a unique identifier for a feed.

```
go
type ID [32]byte
```

Methods

- FromString: Converts a string into an ID.

```
go
func (f ID) FromString(s string) (err error)
```

- MarshalJSON: Serializes the ID into JSON.

```
go
func (f ID) MarshalJSON() (b []byte, err error)
```

- String: Returns the string representation of the ID.

```
go
func (f ID) String() (id string)
```


- UnmarshalJSON: Deserializes the ID from JSON.

```
go
func (f ID) UnmarshalJSON(b []byte) (err error)
```

- Version: Returns the version of the feed.

```
go
func (f ID) Version() FeedVersion
```

report

Installation

```
go
import report "github.com/smartcontractkit/data-streams-sdk/go/report"
```

Types

Data interface

Interface that represents the actual report data and attributes.

```
go
type Data interface {
    v1.Data | v2.Data | v3.Data
    Schema() abi.Arguments
}
```

Report struct

Represents the report envelope that contains the full report payload, its Feed ID, and timestamps.

```
go
type Report[T Data] struct {
    Data          T
    ReportContext [3][32]byte
    ReportBlob    []byte
    RawRs         []byte
    RawSs         []byte
    RawVs         [32]byte
}
```

Functions

Decode

Decodes the report serialized bytes and its data.

```
go
func DecodeT Data (r Report[T], err error)
```

Example:

```
go
payload, := hex.DecodeString(
```


REST endpoint to query specific reports	https://api.testnet-dataengine.chain.link https://api.dataengine.chain.link
---	---

Authentication

Headers

All routes require the following three headers for user authentication:

Header	Description
Authorization	The user's unique identifier, provided as a UUID (Universally Unique Identifier).
X-Authorization-Timestamp	The current timestamp, with precision up to milliseconds. The timestamp must closely synchronize with the server time, allowing a maximum discrepancy of 5 seconds (by default).
X-Authorization-Signature-SHA256	The HMAC (Hash-based Message Authentication Code) signature, generated by hashing parts of the request and its metadata using SHA-256 with a shared secret key.

API endpoints

Return a single report at a given timestamp

Endpoint

/api/v1/reports

Type	Description	Parameter(s)
HTTP GET	Returns a single report for a given timestamp.	<ul style="list-style-type: none"> feedID: A Data Streams feed ID. timestamp: The Unix timestamp for the report.

Sample request

```
http
GET /api/v1/reports?feedID=<feedID>&timestamp=<timestamp>
```

Sample response

```
json
{
  "report": {
    "feedID": "Hex encoded feedId.",
    "validFromTimestamp": "Report's earliest applicable timestamp (in seconds).",
    "observationsTimestamp": "Report's latest applicable timestamp (in seconds).",
    "fullReport": "A blob containing the report context and body. Encode the fee token into the payload before passing it to the contract for verification."
  }
}
```

Return a single report with the latest timestamp

Endpoint

/api/v1/reports/latest

Type	Parameter(s)
HTTP GET	feedID: A Data Streams feed ID.

Sample request

```
http
GET /api/v1/reports/latest?feedID=<feedID>
```

Sample response

```
json
{
  "report": {
    "feedID": "Hex encoded feedId.",
    "validFromTimestamp": "Report's earliest applicable timestamp (in
seconds).",
    "observationsTimestamp": "Report's latest applicable timestamp (in
seconds).",
    "fullReport": "A blob containing the report context and body. Encode the fee
token into the payload before passing it to the contract for verification."
  }
}
```

Return a report for multiple FeedIDs at a given timestamp

Endpoint

/api/v1/reports/bulk

Type	Description
Parameter(s)	
HTTP GET	Return a report for multiple FeedIDs at a given timestamp.
feedIDs: A comma-separated list of Data Streams feed IDs.timestamp: The Unix timestamp for the reports.	

Sample request

```
http
GET /api/v1/reports/bulk?feedIDs=<FeedID1>,<FeedID2>,...&timestamp=<timestamp>
```

Sample response

```
json
{
  "reports": [
    {
      "feedID": "Hex encoded feedId.",
      "validFromTimestamp": "Report's earliest applicable timestamp (in
```


Code) signature, generated by hashing parts of the request and its metadata using SHA-256 with a shared secret key. |

WebSocket Connection

Establish a streaming WebSocket connection that sends reports for the given feedID(s) after they are verified.

Endpoint

/api/v1/ws

Type	Parameter(s)
WebSocket	feedIDs: A comma-separated list of Data Streams feed IDs.

Sample request

```
http
GET /api/v1/ws?feedIDs=<feedID1>,<feedID2>,...
```

Sample response

```
json
{
  "report": {
    "feedID": "hex encoded feedId",
    "fullReport": "a blob containing the report context + body, can be passed
unmodified to the contract for verification"
  }
}
```

Error response codes

Status Code	Description
-----	-----
-----	-----
-----	-----
400 Bad Request	This error is triggered when: There is any missing/malformed query argument.Required headers are missing or provided with incorrect values.
401 Unauthorized User	This error is triggered when: Authentication fails, typically because the HMAC signature provided by the client doesn't match the one expected by the server.A user requests access to a feed without the appropriate permission or that does not exist.
500 Internal Server	Indicates an unexpected condition encountered by the server, preventing it from fulfilling the request. This error typically points to issues on the server side.

streams-direct-onchain-verification.mdx:

```
---
section: dataStreams
date: Last Modified
title: "Onchain Data Verification"
---
```

```
import { CodeSample, Aside } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<Aside type="caution" title="Onchain Data Verification">
```

Onchain verification ensures the integrity of reports by confirming their authenticity as signed by the Decentralized

Oracle Network (DON). It is the responsibility of the application contract(s) to determine the suitability of the report data for any further actions, such as trade execution.

```
</Aside>
```

```
<DataStreams section="dsNotes" />
```

Verify reports onchain

To verify data onchain, Streams Direct requires several interfaces.

The primary onchain interaction occurs between the IVerifierProxy interface and your protocol's client contract. Find the Verifier proxy address for each feed on the Data Streams Feed IDs page.

Interfaces

- IVerifierProxy
- IFeeManager

In the current code example for verifying reports onchain using Streams Direct, these interfaces are specified in the example itself. Imports for these interfaces will be available in the future.

Contract example to verify report data onchain

This contract example allows you to verify reports and pay the verification fee in LINK tokens. Your contract must have sufficient LINK tokens to pay for the verification fee. Learn how to fund your contract with LINK tokens.

```
<DataStreams section="asideDisclaimer" />
```

```
<CodeSample src="samples/DataStreams/ClientReportsVerifier.sol" />
```

index.mdx:

```
---
section: dataStreams
date: Last Modified
title: "Streams Direct guides"
isIndex: true
---
```

Explore several guides to learn how to use the Streams Direct implementation with the Data Streams SDK for Go.

- Fetch and decode reports: Learn how to fetch and decode reports from the Data Streams Aggregation Network.
- Stream and decode reports (WebSocket): Learn how to listen for real-time reports from the Data Streams Aggregation Network, decode the report data, and log their attributes.
- Verify report data onchain: Learn how to verify onchain the integrity of reports by confirming their authenticity as signed by the Decentralized Oracle Network (DON).


```
# streams-direct-api.mdx:
```

```
---
```

```
section: dataStreams
```

```
date: Last Modified
```

```
title: "Fetch and decode reports using the SDK"
```

```
whatsnext:
```

```
{
  "Learn how to stream and decode reports via a WebSocket connection": "/data-
streams/tutorials/streams-direct/streams-direct-ws",
  "Learn how to verify your data onchain": "/data-streams/reference/streams-
direct/streams-direct-onchain-verification",
  "Find the list of available Data Streams Feed IDs": "/data-streams/stream-
ids",
}
```

```
---
```

```
import { CopyText } from "@components"
```

```
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<DataStreams section="dsNotes" />
```

In this guide, you'll learn how to use Chainlink Data Streams with the Streams Direct implementation and the Data Streams SDK for Go to fetch and decode reports from the Data Streams Aggregation Network. You'll set up your Go project, retrieve reports, decode them, and log their attributes.

```
<DataStreams section="asideDisclaimer" />
```

Requirements

- Git: Make sure you have Git installed. You can check your current version by running `<CopyText text="git --version" code/>` in your terminal and download the latest version from the official Git website if necessary.
- Go Version: Make sure you have Go version 1.21 or higher. You can check your current version by running `go version` in your terminal and download the latest version from the official Go website if necessary.
- API Credentials: Access to the Streams Direct implementation requires API credentials. If you haven't already, contact us to talk to an expert about integrating Chainlink Data Streams with your applications.

Guide

You'll start with the set up of your Go project. Next, you'll fetch and decode reports for both single and multiple feeds, and log their attributes to your terminal.

Set up your Go project

1. Create a new directory for your project and navigate to it:

```
bash
mkdir my-data-streams-project
cd my-data-streams-project
```

1. Initialize a new Go module:

```
bash
go mod init my-data-streams-project
```

1. Install the Data Streams SDK:

```
bash
go get github.com/smartcontractkit/data-streams-sdk/go
```

Fetch and decode a report with a single feed

1. Create a new Go file, single-feed.go, in your project directory:

```
bash
touch single-feed.go
```

1. Insert the following code example and save your single-feed.go file:

```
go
package main

import (
    "context"
    "fmt"
    "os"
    "time"

    streams "github.com/smartcontractkit/data-streams-sdk/go"
    feed "github.com/smartcontractkit/data-streams-sdk/go/feed"
    report "github.com/smartcontractkit/data-streams-sdk/go/report"
    v3 "github.com/smartcontractkit/data-streams-sdk/go/report/v3"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Printf("Usage: go run main.go [FeedID]\n")
        os.Exit(1)
    }
    feedIDInput := os.Args[1]

    // Define the configuration for the SDK client.
    cfg := streams.Config{
        ApiKey: "YOURAPIKEY",
        ApiSecret: "YOURAPISECRET",
        RestURL: "https://api.testnet-dataengine.chain.link",
        Logger: streams.LogPrintf,
    }

    // Initialize the SDK client.
    client, err := streams.New(cfg)
    if err != nil {
        cfg.Logger("Failed to create client: %v\n", err)
        os.Exit(1)
    }

    // Create context for timeout.
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()

    // Define the FeedID from user input.
    var FeedID feed.ID
    if err := FeedID.FromString(feedIDInput); err != nil {
        cfg.Logger("Failed to parse feed ID: %v\n", err)
        os.Exit(1)
    }

    // Fetch the latest report for the given FeedID.
    reportResponse, err := client.GetLatestReport(ctx, FeedID)
```

```

if err != nil {
    cfg.Logger("Failed to get latest report: %v\n", err)
    os.Exit(1)
}

// Log the contents of the report before decoding
cfg.Logger("Raw report data: %+v\n", reportResponse)

// Decode the report.
decodedReport, err := report.Decodev3.Data
if err != nil {
    cfg.Logger("Failed to decode report: %v\n", err)
    os.Exit(1)
}

// Log the decoded report
cfg.Logger("\nDecoded Report for Feed ID %s:\n-----\n"+
    "Observations Timestamp: %d\n"+
    "Benchmark Price      : %s\n"+
    "Bid                  : %s\n"+
    "Ask                  : %s\n"+
    "Valid From Timestamp : %d\n"+
    "Expires At           : %d\n"+
    "Link Fee              : %s\n"+
    "Native Fee            : %s\n"+
    "-----\n",
    feedIDInput,
    decodedReport.Data.ObservationsTimestamp,
    decodedReport.Data.BenchmarkPrice.String(),
    decodedReport.Data.Bid.String(),
    decodedReport.Data.Ask.String(),
    decodedReport.Data.ValidFromTimestamp,
    decodedReport.Data.ExpiresAt,
    decodedReport.Data.LinkFee.String(),
    decodedReport.Data.NativeFee.String(),
)
}

```

1. Download the required dependencies and update the go.mod and go.sum files:

```

bash
go mod tidy

```

1. Set up the SDK client configuration within single-feed.go with your API credentials:

```

go
cfg := streams.Config{
    ApiKey:    "YOURAPIKEY",
    ApiSecret: "YOURAPISECRET",
    RestURL:   "https://api.testnet-dataengine.chain.link",
    Logger:    streams.LogPrintf,
}

```

- Replace ApiKey and ApiSecret with your API credentials.
- RestURL is the REST endpoint to poll for specific reports. See the Streams Direct Interface page for more information.

See the SDK Reference page for more configuration options.

1. For this example, you will read from the ETH/USD Data Streams feed. This feed ID is <CopyText
text="0x000359843a543ee2fe414dc14c7e7920ef10f4372990b79d6361cdc0dd1ba782"
code/>. See the Data Streams Feed IDs page for a complete list of available
assets.

Execute your application:

```
bash
go run single-feed.go
0x000359843a543ee2fe414dc14c7e7920ef10f4372990b79d6361cdc0dd1ba782
```

Expect output similar to the following in your terminal:

[illegible]Decoded Report for Feed ID
0x000359843a543ee2fe414dc14c7e7920ef10f4372990b79d6361cdc0dd1ba782:

```
Observations Timestamp: 1722448043
Benchmark Price       : 3318947247532739300000
Bid                   : 3318897160259676600000
Ask                   : 3319031900000000000000
Valid From Timestamp  : 1722448043
Expires At            : 1722534443
Link Fee               : 7633426300389000
Native Fee             : 30130035984900
```

Decoded report details

The decoded report details include:

Attribute	Value
Description	


```

----- |
| Feed ID |
0x000359843a543ee2fe414dc14c7e7920ef10f4372990b79d6361cdc0dd1ba782 | The unique
identifier for the Data Streams feed. In this example, the feed is for ETH/USD.
|
| Observations Timestamp | 1722448043
| The timestamp indicating when the data was captured.
|
| Benchmark Price | 3318947247532739300000
| The observed price in the report, with 18 decimals. For readability:
3,318.9472475327393 USD per ETH.
|
| Bid | 3318897160259676600000
| The highest price a buyer is willing to pay for an asset, with 18 decimals.
For readability: 3,318.8971602596766 USD per ETH. Learn more about the Bid
price. |
| Ask | 331903190000000000000000
| The lowest price a seller is willing to accept for an asset, with 18 decimals.
For readability: 3,319.03190000000000 USD per ETH. Learn more about the Ask
price. |
| Valid From Timestamp | 1722448043
| The start validity timestamp for the report, indicating when the data becomes
relevant.
|
| Expires At | 1722534443
| The expiration timestamp of the report, indicating the point at which the data
becomes outdated.
|
| Link Fee | 7633426300389000
| The fee to pay in LINK tokens for the onchain verification of the report data.
With 18 decimals. For readability: 0.007633426300389 LINK.
|
| Native Fee | 30130035984900
| The fee to pay in the native blockchain token (e.g., ETH on Ethereum) for the
onchain verification of the report data. With 18 decimals. Note: This example
fee is not indicative of actual fees. |

```

Payload for onchain verification

In this guide, you log and decode the fullReport payload to extract the report data. In a production environment, you should verify the data onchain to ensure its integrity and authenticity. Refer to the [Verify report data onchain guide](#).

Fetch and decode reports for multiple feeds

1. Create a new Go file, multiple-feeds.go, in your project directory:

```

bash
touch multiple-feeds.go

```

1. Insert the following code example in your multiple-feeds.go file:

```

go
package main

import (
    "context"
    "fmt"
    "os"
    "time"

```

```

streams "github.com/smartcontractkit/data-streams-sdk/go"
feed "github.com/smartcontractkit/data-streams-sdk/go/feed"
report "github.com/smartcontractkit/data-streams-sdk/go/report"
v3 "github.com/smartcontractkit/data-streams-sdk/go/report/v3"
)

func main() {
    if len(os.Args) < 3 { // Ensure there are at least two Feed IDs provided
        fmt.Printf("Usage: go run multiple-feeds.go [FeedID1] [FeedID2] ...\n"
n")
        os.Exit(1)
    }

    // Define the configuration for the SDK client
    cfg := streams.Config{
        ApiKey: "YOURAPIKEY",
        ApiSecret: "YOURAPISECRET",
        RestURL: "https://api.testnet-dataengine.chain.link",
        Logger: streams.LogPrintf,
    }

    // Initialize the SDK client
    client, err := streams.New(cfg)
    if err != nil {
        cfg.Logger("Failed to create client: %v\n", err)
        os.Exit(1)
    }

    // Create context for timeout
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()

    // Parse Feed IDs from command line arguments
    var ids []feed.ID
    for , arg := range os.Args[1:] {
        var fid feed.ID
        if err := fid.FromString(arg); err != nil {
            cfg.Logger("Failed to parse feed ID %s: %v\n", arg, err)
            os.Exit(1)
        }
        ids = append(ids, fid)
    }

    // Fetch reports for the given feed IDs
    reportResponses, err := client.GetReports(ctx, ids,
uint64(time.Now().Unix())) // Using current time as an example timestamp
    if err != nil {
        cfg.Logger("Failed to get reports: %v\n", err)
        os.Exit(1)
    }

    for , reportResponse := range reportResponses {
        // Log the contents of the report before decoding
        cfg.Logger("Raw report data for Feed ID %s: %v\n",
reportResponse.FeedID.String(), reportResponse)

        // Decode the report
        decodedReport, err := report.Decodev3.Data
        if err != nil {
            cfg.Logger("Failed to decode report for Feed ID %s: %v\n",
reportResponse.FeedID, err)
            continue // Skip to next report if decoding fails
        }
    }
}

```

```
// Log the decoded report
cfg.Logger("\nDecoded Report for Feed ID %s:\n-----\n"+
    "Observations Timestamp: %d\n"+
    "Benchmark Price       : %s\n"+
    "Bid                   : %s\n"+
    "Ask                   : %s\n"+
    "Valid From Timestamp  : %d\n"+
    "Expires At            : %d\n"+
    "Link Fee              : %s\n"+
    "Native Fee            : %s\n"+
    "-----\n",
    reportResponse.FeedID.String(),
    decodedReport.Data.ObservationsTimestamp,
    decodedReport.Data.BenchmarkPrice.String(),
    decodedReport.Data.Bid.String(),
    decodedReport.Data.Ask.String(),
    decodedReport.Data.ValidFromTimestamp,
    decodedReport.Data.ExpiresAt,
    decodedReport.Data.LinkFee.String(),
    decodedReport.Data.NativeFee.String(),
)
}
```

1. Replace `ApiKey` and `ApiSecret` with your API credentials in the SDK client configuration:

```
go
cfg := streams.Config{
    ApiKey:    "YOURAPIKEY",
    ApiSecret: "YOURAPISecret",
    RestURL:   "https://api.testnet-dataengine.chain.link",
    Logger:    streams.LogPrintf,
}
```

1. For this example, you will read from the ETH/USD and LINK/USD Data Streams feeds. Run your application:

```
bash
go run multiple-feeds.go
0x000359843a543ee2fe414dc14c7e7920ef10f4372990b79d6361cdc0dd1ba782
0x00036fe43f87884450b4c7e093cd5ed99cac6640d8c2000e6afc02c8838d0265
```

Expect to see the output below in your terminal:

[illegible]

[illegible]

```
Observations Timestamp: 1722448966
Benchmark Price       : 3310471484260632700000
Bid                   : 3310433423677103300000
Ask                   : 3310509544844162000000
Valid From Timestamp  : 1722448966
Expires At            : 1722535366
Link Fee              : 7666787759463600
Native Fee            : 30207177580400
```

Decoded Report for Feed ID
0x00036fe43f87884450b4c7e093cd5ed99cac6640d8c2000e6afc02c8838d0265:

```
Observations Timestamp: 1722448966
Benchmark Price       : 1304825600000000000000
Bid                   : 13044134801824773000
Ask                   : 1305075900000000000000
Valid From Timestamp  : 1722448966
Expires At            : 1722535366
Link Fee              : 7663859446044000
Native Fee            : 30206947453900
```


In this guide, you log and decode the fullReport payloads to extract the report data. In a production environment, you should verify the data onchain to ensure its integrity and authenticity. Refer to the [Verify report data onchain guide](#).

Explanation

Fetching reports

Your application uses the Data Streams SDK to fetch reports from the Data Streams Aggregation Network. Different functions from the SDK are invoked based on whether a single feed or multiple feeds are queried:

- **Single Feed:** The application calls the `GetLatestReport` function from the SDK's streams package to fetch a single feed report. This function sends an API request to retrieve the latest available report for a specified feed ID. The response includes the report's data in a structured format suitable for further processing and analysis.

- **Multiple Feeds:** The `GetReports` function is used for multiple feeds. This function allows you to fetch reports for multiple feed IDs simultaneously, specifying a timestamp to retrieve reports valid at a particular point in time.

The responses include a fullReport blob containing the encoded report's context and observations.

Decoding reports

After retrieving the report data, the application decodes it using the `Decode` function available in the report package of the SDK. This function is designed to handle different data formats by specifying the version of the data structure as a type parameter (e.g., `v3.Data`).

The `Decode` function unpacks the encoded report blob into a usable data structure (`Report[v3.Data]`). It extracts information such as observation timestamps, benchmark prices, bid and ask prices from the report data.

Handling the decoded data

The application logs the report data to the terminal. However, this data can be used for further processing, analysis, or display in your own application.

```
# streams-direct-onchain-verification.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Verify report data onchain"
whatsnext: { "Find the list of available Data Streams Feed IDs": "/data-streams/stream-ids" }
---

import { CodeSample, CopyText, ClickToZoom } from "@components"
import { Tabs } from "@components/Tabs"
import DataStreams from "@features/data-streams/common/DataStreams.astro"

<DataStreams section="dsNotes" />
```

In this tutorial, you'll learn how to verify onchain the integrity of reports by confirming their authenticity as signed by the Decentralized Oracle Network (DON). You'll use a verifier contract to verify the data onchain and pay the

verification fee in LINK tokens.

```
<DataStreams section="asideDisclaimer" />
```

Before you begin

Make sure you understand how to use the Streams Direct implementation of Chainlink Data Streams to fetch reports via the REST API or WebSocket connection. Refer to the following guides for more information:

- Fetch and decode reports via a REST API
- Stream and decode reports via WebSocket

Requirements

- This guide requires testnet ETH and LINK on Arbitrum Sepolia. Both are available at faucets.chain.link.
- Learn how to Fund your contract with LINK.

Tutorial

Deploy the verifier contract

Deploy a ClientReportsVerifier contract on Arbitrum Sepolia. This contract is enabled to verify reports and pay the verification fee in LINK tokens.

1. Open the ClientReportsVerifier.sol contract in Remix.

```
<CodeSample src="samples/DataStreams/ClientReportsVerifier.sol"
showButtonOnly />
```

1. Select the ClientReportsVerifier.sol contract in the Solidity Compiler tab.

```
<ClickToZoom
src="/images/data-streams/onchain-verification/solidity-compiler.webp"
alt="Chainlink Data Streams - Verify Report Data Onchain - Solidity
Compiler"
style="max-width: 70%;"
/>
```

1. Compile the contract.

1. Open MetaMask and set the network to Arbitrum Sepolia. If you need to add Arbitrum Sepolia to your wallet, you can find the chain ID and the LINK token contract address on the LINK Token Contracts page.

```
- <a
  class="erc-token-address"
  id="4216140xb1D4538B4571d411F07960EF2838Ce337FE1E80E"
  href="/resources/link-token-contracts#arbitrum-sepolia-testnet"
>
  Arbitrum Sepolia testnet and LINK token contract
</a>
```

1. On the Deploy & Run Transactions tab in Remix, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Arbitrum Sepolia.

```
<ClickToZoom
src="/images/data-streams/onchain-verification/injected-provider.webp"
alt="Chainlink Data Streams - Verify Report Data Onchain - Injected
Provider MetaMask"
style="max-width: 70%;"
/>
```


1f06d7475c2e207867f53cb4d0fb7387880109b3a2192b1b4027cce218afeeb5b2b2110a9bfac8e4a976f5e2c5e11e08afceafda9a8e13aa99

```
{ " " }  
<ClickToZoom
```

```
src="/images/data-streams/onchain-verification/deployed-  
clientReportsVerifier-2.webp"  
alt="Chainlink Data Streams Remix Deployed ClientReportsVerifier Contract"  
style="max-width: 70%;"  
/>
```

1. Click the verifyReport button to call the function. MetaMask prompts you to accept the transaction.

1. Click the lastDecodedPrice getter function to view the decoded price from the verified report. The answer on the ETH/USD feed uses 18 decimal places, so an answer of 3257579704051546000000 indicates an ETH/USD price of 3,257.579704051546. Each Data Streams feed uses a different number of decimal places for answers. See the Data Streams Feed IDs page for more information.

```
<ClickToZoom  
src="/images/data-streams/onchain-verification/price-from-verified-  
report.webp"  
alt="Chainlink Data Streams - Price from Verified Report"  
style="max-width: 70%;"  
/>
```

Examine the code

The example code you deployed has all the interfaces and functions required to verify Data Streams reports onchain.

```
<CodeSample src="samples/DataStreams/ClientReportsVerifier.sol" />
```

Initializing the contract

When deploying the contract, you define the verifier proxy address for the Data Streams feed you want to read from. You can find this address on the Data Streams Feed IDs page. The verifier proxy address provides functions that are required for this example:

- The sfeeManager function to estimate the verification fees.
- The verify function to verify the report onchain.

Verifying a report

The verifyReport function is the main function of the contract that verifies the report onchain. It follows the steps below:

- Fee calculation: It interacts with the FeeManager contract, accessible via the verifier proxy contract, to determine the fees associated with verifying the report.
- Token approval: It grants approval to the rewardManager contract to spend the required amount of LINK tokens from its balance.
- Report verification: The core verification step involves calling the verify function from the verifier proxy contract. This function takes the (unverified) report payload and the encoded fee token address as inputs and returns the verified report data.
- Data decoding and storage: In this example, the verified report data is

decoded into a Report struct, extracting the report data. The extracted price data is then emitted through the DecodedPrice event and stored in the lastdecodedprice state variable.

```
# streams-direct-ws.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Stream and decode reports via WebSocket using the SDK"
whatsnext:
  {
    "Learn how to verify your data onchain": "/data-streams/reference/streams-
direct/streams-direct-onchain-verification",
    "Find the list of available Data Streams Feed IDs": "/data-streams/stream-
ids",
  }
---
```

```
import { CopyText } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"
```

```
<DataStreams section="dsNotes" />
```

In this guide, you'll learn how to use Chainlink Data Streams with the Streams Direct implementation and the Data Streams SDK for Go to subscribe to real-time reports via a WebSocket connection. You'll set up your Go project, listen for real-time reports from the Data Streams Aggregation Network, decode the report data, and log their attributes to your terminal.

```
<DataStreams section="asideDisclaimer" />
```

Requirements

- Git: Make sure you have Git installed. You can check your current version by running `<CopyText text="git --version" code/>` in your terminal and download the latest version from the official Git website if necessary.
- Go Version: Make sure you have Go version 1.21 or higher. You can check your current version by running `go version` in your terminal and download the latest version from the official Go website if necessary.
- API Credentials: Access to the Streams Direct implementation requires API credentials. If you haven't already, contact us to talk to an expert about integrating Chainlink Data Streams with your applications.

Guide

Set up your Go project

1. Create a new directory for your project and navigate to it:

```
bash
mkdir my-data-streams-project
cd my-data-streams-project
```

1. Initialize a new Go module:

```
bash
go mod init my-data-streams-project
```

1. Install the Data Streams SDK:

```
bash
go get github.com/smartcontractkit/data-streams-sdk/go
```

Establish a WebSocket connection and listen for real-time reports

1. Create a new Go file, stream-feeds.go, in your project directory:

```
bash
touch stream-feeds.go
```

1. Insert the following code example and save your stream-feeds.go file:

```
go
package main

import (
    "context"
    "fmt"
    "os"
    "time"

    streams "github.com/smartcontractkit/data-streams-sdk/go"
    feed "github.com/smartcontractkit/data-streams-sdk/go/feed"
    report "github.com/smartcontractkit/data-streams-sdk/go/report"
    v3 "github.com/smartcontractkit/data-streams-sdk/go/report/v3"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Println("Usage: go run stream-feeds.go [FeedID1] [FeedID2] ...")
        os.Exit(1)
    }

    // Set up the SDK client configuration
    cfg := streams.Config{
        ApiKey: "YOURAPIKEY",
        ApiSecret: "YOURAPISECRET",
        WsURL: "wss://ws.testnet-dataengine.chain.link",
        Logger: streams.LogPrintf,
    }

    // Create a new client
    client, err := streams.New(cfg)
    if err != nil {
        cfg.Logger("Failed to create client: %v\n", err)
        os.Exit(1)
    }

    // Parse the feed IDs from the command line arguments
    var ids []feed.ID
    for , arg := range os.Args[1:] {
        var fid feed.ID
        if err := fid.FromString(arg); err != nil {
            cfg.Logger("Invalid feed ID %s: %v\n", arg, err)
            os.Exit(1)
        }
        ids = append(ids, fid)
    }

    ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
    defer cancel()
```

```

// Subscribe to the feeds
stream, err := client.Stream(ctx, ids)
if err != nil {
    cfg.Logger("Failed to subscribe: %v\n", err)
    os.Exit(1)
}

defer stream.Close()
for {
    reportResponse, err := stream.Read(context.Background())
    if err != nil {
        cfg.Logger("Error reading from stream: %v\n", err)
        continue
    }

    // Log the contents of the report before decoding
    cfg.Logger("Raw report data: %v\n", reportResponse)

    // Decode each report as it comes in
    decodedReport, decodeErr := report.Decodev3.Data
    if decodeErr != nil {
        cfg.Logger("Failed to decode report: %v\n", decodeErr)
        continue
    }

    // Log the decoded report
    cfg.Logger("\n--- Report Feed ID: %s ---\n" +
        "-----\n" +
        "Observations Timestamp : %d\n" +
        "Benchmark Price       : %s\n" +
        "Bid                   : %s\n" +
        "Ask                   : %s\n" +
        "Valid From Timestamp  : %d\n" +
        "Expires At           : %d\n" +
        "Link Fee              : %s\n" +
        "Native Fee           : %s\n" +
        "-----\n",
        reportResponse.FeedID.String(),
        decodedReport.Data.ObservationsTimestamp,
        decodedReport.Data.BenchmarkPrice.String(),
        decodedReport.Data.Bid.String(),
        decodedReport.Data.Ask.String(),
        decodedReport.Data.ValidFromTimestamp,
        decodedReport.Data.ExpiresAt,
        decodedReport.Data.LinkFee.String(),
        decodedReport.Data.NativeFee.String(),
    )

    // Also, log the stream stats
    cfg.Logger("\n--- Stream Stats ---\n" +
        stream.Stats().String() + "\n" +
        "-----\n",
    )
}
}

```

1. Download the required dependencies and update the go.mod and go.sum files:

```

bash
go mod tidy

```

```
go
cfg := streams.Config{
    ApiKey: "YOURAPIKEY",
    ApiSecret: "YOURAPISECRET",
    WsURL: "wss://ws.testnet-dataengine.chain.link",
    Logger: streams.LogPrintf,
}
```

- See the [SDK Reference](#) page for more configuration options.

Execute your application:

Expect output similar to the following in your terminal:

2024-07-31T15:34:27-05:00

```
0x000359843a543ee2fe414dc14c7e7920ef10f4372990b79d6361cdc0dd1ba782 - - -
```

```
Observations Timestamp : 1722458067
Benchmark Price        : 3232051369848762000000
```


[illegible]


```

| Benchmark Price          | 323277310000000000000000
| The observed price in the report, with 18 decimals. For readability:
3,232.77310000000000 USD per ETH.
|
| Bid                      | 323272870000000000000000
| The highest price a buyer is willing to pay for an asset, with 18 decimals.
For readability: 3,232.72870000000000 USD per ETH. Learn more about the Bid
price. |
| Ask                      | 323281740000000000000000
| The lowest price a seller is willing to accept for an asset, with 18 decimals.
For readability: 3,232.81740000000000 USD per ETH. Learn more about the Ask
price. |
| Valid From Timestamp    | 1722458069
| The start validity timestamp for the report, indicating when the data becomes
relevant.
|
| Expires At              | 1722544469
| The expiration timestamp of the report, indicating the point at which the data
becomes outdated.
|
| Link Fee                 | 7775942157527100
| The fee to pay in LINK tokens for the onchain verification of the report data.
With 18 decimals. For readability: 0.0077759421575271 LINK.
|
| Native Fee               | 30933194785600
| The fee to pay in the native blockchain token (e.g., ETH on Ethereum) for the
onchain verification of the report data. With 18 decimals. Note: This example
fee is not indicative of actual fees. |

```

Payload for onchain verification

In this guide, you log and decode the fullReport payload to extract the report data. In a production environment, you should verify the data onchain to ensure its integrity and authenticity. Refer to the [Verify report data onchain guide](#).

Explanation

Establishing a WebSocket connection and listening for reports

Your application uses the Stream function in the Data Streams SDK's client package to establish a real-time WebSocket connection with the Data Streams Aggregation Network.

Once the WebSocket connection is established, your application subscribes to one or more Data Streams feeds by passing an array of feed.IDs to the Stream function. This subscription lets the client receive real-time updates whenever new report data is available for the specified feeds.

Decoding a report

As data reports arrive via the established WebSocket connection, they are processed in real-time:

- Reading streams: The Read method on the returned Stream object is continuously called within a loop. This method blocks until new data is available, ensuring that all incoming reports are captured as soon as they are broadcasted.
- Decoding reports: For each received report, the SDK's Decode function parses and transforms the raw data into a structured format (v3.Data). This decoded data includes data such as the benchmark price and bid and ask prices.

Handling the decoded data

In this example, the application logs the structured report data to the terminal. However, this data can be used for further processing, analysis, or display in your own application.

```
# index.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Streams Trade guides"
isIndex: true
---
```

Explore several guides to learn how to use the Streams Trade implementation of Data Streams.

- Getting Started: Learn how to read data from a Data Streams feed, verify the answer onchain, and store it.
- Handle StreamsLookup errors: Learn how to handle potential errors or edge cases in StreamsLookup upkeeps.

```
# streams-trade-lookup-error-handler.mdx:
```

```
---
section: dataStreams
date: Last Modified
title: "Using the StreamsLookup error handler"
metadata:
  linkTOWallet: true
whatsnext: {
  "Find the list of available stream IDs.": "/data-streams/stream-ids",
  "Find the schema of data to expect from Data Streams reports.":
    "/data-streams/reference/report-schema",
  "Learn more about Log Trigger upkeeps": "/chainlink-automation/guides/log-trigger/",
}
---
```

```
import { Aside } from "@components"
import DataStreams from "@features/data-streams/common/DataStreams.astro"

<Aside type="note" title="Mainnet Access">
  Chainlink Data Streams is available on Arbitrum Mainnet and Arbitrum Sepolia.
</Aside>

<Aside type="note" title="Talk to an expert">
  <a href="https://chainlinkcommunity.typeform.com/datastreams?#refid=docs">Contact us</a> to talk to an expert about
  integrating Chainlink Data Streams with your applications.
</Aside>

<DataStreams section="streamsLookupErrorHandler" />
```

```
# conceptual-overview.mdx:
```

```
---
section: global
date: Last Modified
title: "Smart Contract Overview"
excerpt: "Smart Contracts and Chainlink"
```

```
whatsnext:
  {
    "Deploy Your First Smart Contract": "/quickstarts/deploy-your-first-
contract",
    "Consuming Data Feeds": "/data-feeds/getting-started",
  }
metadata:
  title: "Conceptual Overview"
  description: "Learn the basic concepts about what smart contracts are and, how
to write them, and how Chainlink oracles work with smart contracts."
  image: "/files/1a63254-link.png"
---
```

```
import { CodeSample } from "@components"
import { YouTube } from "@astro-community/astro-embed-youtube"
```

Welcome to the Smart Contract Getting Started guide. This overview explains the basic concepts of smart contract development and oracle networks.

Skip ahead:

To get your hands on the code right away, you can skip this overview:

- Deploy Your First Smart Contract: If you are new to smart contracts, deploy your first smart contract in an interactive web development environment.
- Learn how to use Data Feeds: If you are already familiar with smart contracts and want to learn how to create hybrid smart contracts, use Chainlink Data Feeds to get asset price data onchain.

```
<YouTube id="https://www.youtube.com/watch?v=rFXSEEQG9YE" />
```

What is a smart contract? What is a hybrid smart contract?

When deployed to a blockchain, a smart contract is a set of instructions that can be executed without intervention from third parties. The smart contract code defines how it responds to input, just like the code of any other computer program.

A valuable feature of smart contracts is that they can store and manage onchain assets (like ETH or ERC20 tokens), just like you can with an Ethereum wallet. Because they have an onchain address like a wallet, they can do everything any other address can. This enables you to program automated actions when receiving and transferring assets.

Smart contracts can connect to real-world market prices of assets to produce powerful applications. Securely connecting smart contracts with offchain data and services is what makes them hybrid smart contracts. This is done using oracles.

What language is a smart contract written in?

The most popular language for writing smart contracts on Ethereum and EVM Chains is Solidity. It was created by the Ethereum Foundation specifically for smart contract development and is constantly being updated. Other languages exist for writing smart contracts on Ethereum and EVM Chains, but Solidity is the language used for Chainlink smart contracts.

If you've ever written Javascript, Java, or other object-oriented scripting languages, Solidity should be easy to understand. Similar to object-oriented languages, Solidity is considered to be a contract-oriented language.

Some networks are not EVM-compatible and use languages other than Solidity for smart contracts:

- Solana

- Writing Solana contracts in Rust
- Writing Solana contracts in C

What does a smart contract look like?

The structure of a smart contract is similar to that of a class in Javascript, with a few differences. For example, the following HelloWorld contract is a simple smart contract that stores a single variable and includes a function to update the value of that variable.

```
<CodeSample src="samples/Tutorials/HelloWorld.sol" />
```

Solidity versions

The first thing that every Solidity file must have is the Solidity version definition. The HelloWorld.sol contract uses version 0.8.7, which is defined in the contract as `pragma solidity 0.8.7;`

You can see the latest versions of the Solidity compiler [here](#). You might also notice smart contracts that are compatible with a range of versions.

```
{/ prettier-ignore /}  
solidity  
pragma solidity >=0.7.0 <0.9.0;
```

This means that the code is written for Solidity version 0.7.0, or a newer version of the language up to, but not including version 0.9.0. The `pragma` selects the compiler, which defines how the code is treated.

Naming a Contract

The `contract` keyword defines the name of the contract, which in this example is `HelloWorld`. This is similar to declaring a class in Javascript. The implementation of `HelloWorld` is inside this definition and denoted with curly braces.

```
{/ prettier-ignore /}  
solidity  
contract HelloWorld {  
  
}
```

Variables

Like Javascript, contracts can have state variables and local variables. State variables are variables with values that are permanently stored in contract storage. The values of local variables, however, are present only until the function is executing. There are also different types of variables you can use within Solidity, such as `string`, `uint256`, etc. Check out the Solidity documentation to learn more about the different kinds of variables and types.

Visibility modifiers are used to define the level of access to these variables. Here are some examples of state variables with different visibility modifiers:

```
{/ prettier-ignore /}  
solidity  
string public message;  
uint256 internal internalVar;  
uint8 private privateVar;
```

Learn more about state variables visibility [here](#).

Constructors

Another familiar concept to programmers is the constructor. When you deploy a contract, the constructor sets the state of the contract when it is first created.

In HelloWorld, the constructor takes in a string as a parameter and sets the message state variable to that string.

```
{/ prettier-ignore /}  
solidity  
constructor(string memory initialMessage) {  
    message = initialMessage;  
}
```

Functions

Functions can access and modify the state of the contract or call other functions on external contracts. HelloWorld has a function named updateMessage, which updates the current message stored in the state.

```
{/ prettier-ignore /}  
solidity  
constructor(string memory initialMessage) {  
    message = initialMessage;  
}  
  
function updateMessage(string memory newMessage) public {  
    message = newMessage;  
}
```

Functions use visibility modifiers to define the access level. Learn more about functions visibility [here](#).

Interfaces

An interface is another concept that is familiar to programmers of other languages. Interfaces define functions without their implementation, which leaves inheriting contracts to define the actual implementation themselves. This makes it easier to know what functions to call in a contract. Here's an example of an interface:

<CodeSample src="samples/Tutorials/Test.sol" />

For this example, override is necessary in the Test contract function because it overrides the base function contained in the numberComparison interface. The contract uses pure instead of view because the isSameNum function in the Test contract does not return a storage variable.

What does "deploying" mean?

Deploying a smart contract is the process of pushing the code to the blockchain, at which point it resides with an onchain address. Once it's deployed, the code cannot be changed and is said to be immutable.

As long as the address is known, its functions can be called through an interface, on Etherscan, or through a library like web3js, web3py, ethers, and more. Contracts can also be written to interact with other contracts on the blockchain.

What is a LINK token?

The LINK token is an ERC677 token that inherits functionality from the ERC20 token standard and allows token transfers to contain a data payload. It is used to pay node operators for retrieving data for smart contracts and also for deposits placed by node operators as required by contract creators.

Any wallet that handles ERC20 tokens can store LINK tokens. The ERC677 token standard that the LINK token implements still retains all functionality of ERC20 tokens.

What are oracles?

Oracles provide a bridge between the real-world and onchain smart contracts by being a source of data that smart contracts can rely on, and act upon.

Oracles play a critical role in facilitating the full potential of smart contract utility. Without a reliable connection to real-world conditions, smart contracts cannot effectively serve the real-world.

How do smart contracts use oracles?

Oracles are most popularly used with Data Feeds. DeFi platforms like AAVE and Synthetix use Chainlink data feed oracles to obtain accurate real-time asset prices in their smart contracts.

Chainlink data feeds are sources of data aggregated from many independent Chainlink node operators. Each data feed has an onchain address and functions that enable contracts to read from that address. For example, the ETH / USD feed.

!Chainlink Feeds List

Smart contracts also use oracles to get other capabilities onchain:

- Generate Verifiable Random Numbers (VRF): Use Chainlink VRF to consume randomness in your smart contracts.
- Call External APIs (Any API): Request & Receive data from any API using the Chainlink contract library.
- Automate Smart Contracts using Chainlink Automation: Automating smart contract functions and regular contract maintenance.

What is Remix?

<YouTube id="https://www.youtube.com/watch?v=JWJWT9cwFbo" />

Remix is a web IDE (integrated development environment) for creating, running, and debugging smart contracts in the browser. It is developed and maintained by the Ethereum foundation. Remix allows Solidity developers to write smart contracts without a development machine since everything required is included in the web interface. It allows for a simplified method of interacting with deployed contracts, without the need for a command line interface. Remix also has support for samples. This means that Remix can load code from Github.

To learn how to use Remix, see the Deploying Your First Smart Contract guide.

```
<div class="remix-callout">
  <a href="/quickstarts/deploy-your-first-contract/">Deploy Your First Smart Contract</a>
</div>
```

What is MetaMask?

Contracts are deployed by other addresses on the network. To deploy a smart

contract, you need an address. Not only that, but you need an address which you can easily use with Remix. Fortunately, MetaMask is just what is needed. MetaMask allows anyone to create an address, store funds, and interact with Ethereum compatible blockchains from a browser extension.

```
# automation-station.mdx:
```

```
---
title: "Automation Station"
description: "Use FlashLiquidity's Automation Station to manage your upkeeps."
image: "QuickStarts-FlashLiquidity-Automation-Station.png"
products: ["automation"]
time: "180 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside, CodeSample, ClickToZoom } from "@components"
```

Overview

This tutorial shows how to create, manage, fund, and batch migrate Chainlink Automation upkeeps using FlashLiquidity's Automation Station.

```
{/ prettier-ignore /}
<div class="remix-callout">
  <a href="https://github.com/flashliquidity/flashliquidity-automation/tree/main">See the code on GitHub</a>
</div>
```

Objective

In this tutorial, you will deploy the AutomationStation contract and register it as an upkeep. This main station upkeep monitors the balances for all your other upkeeps and funds them when the balances are too low. The repository includes Hardhat tasks to help you run each AutomationStation function from the command line.

```
<Aside type="caution" title="Disclaimer">
```

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how

to interact with Chainlink's systems and services so that you can integrate them into your own. This template is

provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key

checks or error handling to make the usage of the product more clear. Do not use the code in this example in a

production environment without completing your own audits and application of best practices. Neither Chainlink Labs,

the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due

to errors in code.

```
</Aside>
```

Before you begin

Before you start this tutorial, complete the following items:

- If you are new to smart contract development, learn how to Deploy Your First Smart Contract.
- Set up a cryptocurrency wallet such as MetaMask.
- To deploy this contract on testnets, ensure the deployer account has testnet ERC-677 LINK and native tokens. Use the Chainlink faucet to retrieve them.

- Install Node.js.
- Install yarn.

Steps to implement

<Accordion title="Install the dependencies" number={1}>

Clone the source and install the required packages:

1. Install Foundry, or update it to the latest version by running foundryup -v in your terminal.

1. Clone the flashliquidity-automation repository:

```
sh
git clone https://github.com/flashliquidity/flashliquidity-automation.git &&
\
cd flashliquidity-automation
```

1. Install the dependencies:

```
sh
yarn setup
```

1. Copy the .env.example file to .env at the root of the directory:

```
sh
cp .env.example .env
```

1. In the .env file, set your private key and the RPC URL for the network you're using.

</Accordion>

<Accordion title="Deploy the AutomationStation contract" number={2}>

Deploy the AutomationStation contract to an Automation-supported network by setting parameters within a deployment script and then running the deployment script.

1. Navigate to the deploy/deployStation.ts file to find the deployment script.

1. Edit the deployment script, filling in the required parameters for the AutomationStation constructor. Make sure to format each value as a string.

At a minimum, you need to fill in these two parameters:

- linkToken: The address of the ERC-677 compatible LINK token. For this example, you can use the address for Sepolia (0x779877A7B0D9E8603169DdbD7836e478b4624789) or see all the LINK token addresses for other networks. Note: If you're using bridged LINK for certain networks, the LINK may not be ERC-677 compatible. Use the Chainlink Pegswap service to convert Chainlink tokens to be ERC-677 compatible before funding the station.
- registrar: The address of the Chainlink Automation registrar for the selected network. For Sepolia, the registrar address is 0xb0E49c5D0d05cbc241d68c05BC5BA1d1B7B72976.

The following parameters have default values in the script. Check that the settings related to refueling upkeeps and minimum balances make sense for your usage, and update them if necessary:

- registerUpkeepSelector: The Chainlink Automation Registrar 4-byte function selector of registerUpkeep, which stays the same across different networks. Leave this at its default value: 0x3f678e11.

- refuelAmount: The amount of LINK tokens that the AutomationStation adds to an underfunded upkeep's balance. (Default value: 1000000000000000000 or 1 LINK)
- stationUpkeepMinBalance: The minimum balance of LINK tokens for the main station upkeep. If the balance falls below this threshold, the main station upkeep is considered underfunded. (Default value: 1000000000000000000 or 1 LINK)
- minDelayNextRefuel: The minimum delay time in seconds between consecutive refuels of the same upkeep (main station upkeep excluded). (Default value: 600 or 10 minutes)
- approveAmountLINK: The initial allowance of LINK tokens to the Chainlink Automation Registrar. (Default value: 1000000000000000000 or 100 LINK)

1. Save your changes to the deploy/deployStation.ts file.
1. Run the deployment script:

```
sh
npx hardhat deploy --network sepolia
```

You can view the supported networks for the deployment scripts inside the Hardhat configuration file (hardhat.config.ts) and add any required network if it is missing.

After you've run the deployment script successfully, the output shows the address of your AutomationStation contract:

```
sh
Compiled 15 Solidity files successfully (evm target: paris).
deploying "AutomationStation" (tx:
0x16fa37c6bdba69ccd8ab585e89d8b5e99045a2b5e97daa2beddc3a7c6859ba31)...: deployed
at 0x084E1fc7411B95a0A5b5833495c4b73E8D7ed3F4 with 2691939 gas
```

</Accordion>

<Accordion title="Fund and initialize the AutomationStation" number={3}>

After deploying the AutomationStation contract, you need to fund the station contract before initializing it as an upkeep. During the initialization process, the main station upkeep is registered. This upkeep monitors all your other registered upkeeps to ensure they are not underfunded; if they are, it adds funds to their balance.

1. Send LINK to the contract address for your deployed AutomationStation. To test this example, you can fund your AutomationStation contract initially with 10 LINK. Get testnet LINK if needed, and see how to fund your contract in MetaMask.

1. When you run the initialize task, it registers your deployed AutomationStation contract as an upkeep on Chainlink Automation. You can pass in the following optional parameters:

- registry: The address of the Chainlink Automation Registry for the selected network. If you set this parameter now, the station's Automation forwarder will be automatically set during initialization. Otherwise, you must set the forwarder before registering other upkeeps. The registry address for Sepolia is 0x86EFBD0b6736Bed994962f9797049422A3A8E8Ad.
- amount: The amount of LINK tokens to fund the main station upkeep. (Default value: 1000000000000000000 or 10 LINK tokens)
- gasLimit: The maximum gas limit that the station upkeep will use for transactions. (Default value: 750000)

If you need to change any other default values for your station upkeep, edit registrationParams in tasks/initializeStation.ts and refer to the registration parameters. By default, the station's contract address is used intentionally for both the adminAddress and upkeepContract fields.

1. Run the initialize task, passing in the Automation registry address for Sepolia, funding the main upkeep with 5 LINK, and setting a gas limit of 500000:

```
sh
npx hardhat initialize --registry 0x86EFBD0b6736Bed994962f9797049422A3A8E8Ad
--amount 5000000000000000000 --gas-limit 500000 --network sepolia
```

After you've run the initialization script successfully, the output shows your AutomationStation upkeep ID:

```
Transaction Hash:
0x3e58940174da3f1474461618ffe1246c2433e0cacd9a80d00eb75297ae7c7226
AutomationStation initialized with upkeep ID:
84877182316128104585193271963895268683397687270564021275663287366641771569317
```

To view your upkeep in the Chainlink Automation App, copy the ID and append it to the URL. For example:

```
https://automation.chain.link/sepolia/
84877182316128104585193271963895268683397687270564021275663287366641771569317
```

Note: If your wallet is connected to the Chainlink Automation App, your AutomationStation upkeep will not appear in the dashboard for the connected wallet, because the owner of the station's upkeep is the address for the deployed AutomationStation contract.

</Accordion>

<Accordion title="Register new upkeeps" number={4}>

After the station is initialized, you can use its registerUpkeep function to register a new Chainlink Automation upkeep. The station monitors the newly registered upkeep to ensure that the upkeep meets the minimum LINK token balance required for execution. If the balance falls below this threshold, the station uses its own LINK token balance to increase the upkeep balance until it meets the required minimum.

To register a new upkeep, run the registerUpkeep Hardhat task.

1. You can pass the following arguments to set the registration parameters for your new upkeep:

Basic configurations

- name: The name of the upkeep displayed in the Chainlink Automation App
- contract: The address of your Automation-compatible contract. If you don't have a deployed Automation-compatible contract on Sepolia, you can use this verified contract: 0x00b68EBB9a12C5bc21ef8E937a3D6E4E963d028D. (View the source code here.)
- gasLimit: The maximum gas limit that the station upkeep will use for transactions
- amount: The LINK token amount to fund the upkeep. (Default value: 5000000000000000000 or 5 LINK)
 - The station's LINK token balance must be greater than or equal to the amount you set in the amount parameter.
 - The allowance of LINK tokens to the Chainlink Automation Registrar must also be greater than or equal to the amount.

Advanced configurations

- checkData: Used to specify static data to pass to your checkUpkeep or checkLog functions in order to execute different code paths. (Default value: 0x)
- triggerType: Used to indicate whether an upkeep is a log trigger upkeep. 0 is Conditional upkeep, 1 is Log trigger upkeep. (Default value: 0)
- triggerConfig: The configuration for your log trigger upkeep. Leave this at 0x for conditional upkeeps. (Default value: 0x)
- offchainConfig: Used to set an optional gas price threshold for your upkeep. Leave this at 0x otherwise. (Default value: 0x)

The default adminAddress is the station's contract address so that the station has permission to manage your new upkeep.

1. Fill in the contract address for your upkeep and run the upkeep registration task:

```
sh
npx hardhat registerUpkeep --name test --contract
0x00b68ebb9a12c5bc21ef8e937a3d6e4e963d028d --gas-limit 1000000 --network sepolia
```

After you've run the upkeep registration script successfully, the output shows your newly registered upkeep ID:

Transaction Hash:

0x56281ea33f37875216e5bd2b473271759945e2b3ec87b7ef2fc7a4899a5e5e10

New upkeep ID:

64045242553038992080465745424964370653102938021228808437683191013349716918956

</Accordion>

<Accordion title="Add existing upkeeps for balance monitoring" number={5}>

Use the addUpkeeps task to add a list of existing upkeeps for your AutomationStation to monitor. This function does not directly modify any of your upkeeps in Chainlink Automation; instead, it adds upkeep IDs to the AutomationStation's supkeepIds watchlist, which is then used for balance monitoring.

1. Prepare the list of upkeep IDs for all the upkeeps that you want to add to the station's watchlist.

1. Run the task:

```
sh
npx hardhat addUpkeeps --network sepolia <upkeepID1> <upkeepID2> <upkeepID3>
... <upkeepIDN>
```

This script outputs a transaction hash:

Transaction Hash:

0x8e8841d6a8dbb0909965e9bd5599308c5120632345211110ef2cc196428a1a14

You can verify that your upkeep IDs were added by checking the value of supkeepIds, or by viewing your deployed station contract in the block explorer to confirm that the addUpkeeps function ran successfully.

</Accordion>

<Accordion title="Manage batches of upkeeps" number={6}>

Unlike the addUpkeeps function, which simply updates the AutomationStation's internal list of upkeeps for balance monitoring, these functions modify batches of upkeeps on Chainlink Automation:

- pauseUpkeeps - Pauses a specified list of upkeeps
- unpauseUpkeeps - Unpauses a specified list of upkeeps
- withdrawUpkeeps - Withdraws LINK from a specified list of canceled upkeeps
- migrateUpkeeps - Migrates a specified list of upkeeps to a newer registry

To use these functions:

1. Prepare the list of upkeep IDs for all the upkeeps that you want to manage.

1. Run the corresponding Hardhat task for the batch function you want to execute on your list of upkeeps. For example, to run pauseUpkeeps.ts:

```
sh
npx hardhat pauseUpkeeps --network sepolia <upkeepID1> <upkeepID2>
<upkeepID3> ... <upkeepIDN>
```

The script outputs a transaction hash:

Transaction Hash:
0x1504119fd7ac19a87aad8a2100145b3bc95cc57a0f56855e28ccdb2984c9e6ab

You can verify that your upkeeps were updated by checking them in the Chainlink Automation App, or by viewing your deployed station contract in the block explorer to confirm that the function ran successfully.

</Accordion>

<Accordion title="Dismantle the Automation Station" number={7}>

1. Remove upkeeps that the Automation Station upkeep is monitoring by using the `removeUpkeep` function. This function removes one upkeep at a time from the station's watchlist. Use the index of the upkeep within the station's watchlist:

```
sh
npx hardhat removeUpkeep --index 1 --network sepolia
```

Alternatively, you can run the `unregisterUpkeep` task to remove individual upkeeps from the station's watchlist and cancel them in Chainlink Automation:

```
sh
npx hardhat unregisterUpkeep --index 1 --network sepolia
```

1. After all upkeeps are removed from the station's watchlist, you can dismantle the station by calling the `dismantle` function:

```
sh
npx hardhat dismantle --network sepolia
```

</Accordion>

batch-reveal.mdx:

```
---
title: "Batch Collection Reveal"
description: "Use Chainlink VRF and Automation in generative art collections."
image: "QuickStarts-Batch-Collection-Reveal.webp"
products: ["automation", "vrf"]
time: "180 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside, CodeSample, ClickToZoom } from "@components"
```

Overview

Using Chainlink VRF in generative art collections is the standard approach for getting provably random source in smart contracts. By batching the reveal process, instead of making VRF calls for each NFT you can save cost up to 100x in a collection of 10,000 with batch size of 100.

This template uses the Chainlink Automation decentralized network to automate the reveal process so you don't have to stand up and maintain an in-house system for automation.

```
{/ prettier-ignore /}  
<div class="remix-callout">  
  <a href="https://github.com/smartcontractkit/chainlink-automation-templates/  
tree/main/batch-nft-reveal">See the code on GitHub</a>  
</div>
```

Objective

In this tutorial, you will use a demo user interface that helps you learn how to automate batch reveal of NFT mints, using Chainlink Automation and VRF. VRF is used to randomize the metadata for the NFTs, and Automation is used to set trigger conditions for when the NFT metadata is revealed via batch size and/or time interval parameters.

The demo includes:

- Setting up and funding a Chainlink VRF subscription
- Automatically generating a smart contract that will then be used by both VRF (as a consumer) and in Automation (as the target contract).

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how

to interact with Chainlink's systems and services so that you can integrate them into your own. This template is

provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key

checks or error handling to make the usage of the product more clear. Do not use the code in this example in a

production environment without completing your own audits and application of best practices. Neither Chainlink Labs,

the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due

to errors in code.

</Aside>

Before you begin

Before you start this tutorial, complete the following items:

- If you are new to smart contract development, learn how to Deploy Your First Smart Contract.
- Set up a cryptocurrency wallet
- To deploy this contract on testnets, ensure the deployer account has testnet ERC-677 LINK. Use the LINK faucet to retrieve testnet LINK.

Steps to implement

<Accordion title="Enter your collection information" number={1}>

1. Go to the demo app and connect to your wallet with the right network. Then click on the Get Started button.

<ClickToZoom

src="/images/automation/qs-batch-reveal/batch-reveal1.png"

alt="Batch Collection Quickstart Step 1"

style="max-width: 70%;"

/>

1. Enter the basic information regarding your collection. You can hover over the question mark icon to learn more about each field.

<ClickToZoom

```
src="/images/automation/qs-batch-reveal/batch-reveal2.png"
alt="Batch Collection Quickstart Step 2"
style="max-width: 70%;"
/>
```

</Accordion>

<Accordion title="Configure your VRF subscription" number={2}>

1. At this stage, you will need to create a VRF subscription and fund it with LINK by visiting the Chainlink VRF Subscription Manager. Follow the steps outlined in the interface to create a subscription. You will receive a confirmation message upon successful creation.

1. Add funds to your subscription. Copy the resulting subscription ID to your clipboard. For this demo, you can add 20 LINK.

```
<ClickToZoom
src="/images/automation/qs-batch-reveal/batch-reveal3.png"
alt="Batch Collection Quickstart Step 3"
style="max-width: 70%;"
/>
```

```
<ClickToZoom
src="/images/automation/qs-batch-reveal/batch-reveal4.png"
alt="Batch Collection Quickstart Step 4"
style="max-width: 70%;"
/>
```

1. Navigate back to the demo app. Paste the VRF Subscription ID into the Subscription ID field and fill out the fields for Queue Size Trigger and Time Trigger (seconds).

1. Upon successful creation, you will see a confirmation page. Copy the address and navigate back to the VRF Subscription Manager.

```
<ClickToZoom
src="/images/automation/qs-batch-reveal/batch-reveal5.png"
alt="Batch Collection Quickstart Step 5"
style="max-width: 70%;"
/>
```

1. In the subscription manager, click Add Consumer and paste the address in the blank field. Click Add Consumer and wait until you receive confirmation. Your consumer should not be added to your subscription. Note: you may have to refresh the page to see the consumer.

```
<ClickToZoom
src="/images/automation/qs-batch-reveal/batch-reveal6.png"
alt="Batch Collection Quickstart Step 6"
style="max-width: 70%;"
/>
```

</Accordion>

<Accordion title="Create an Automation Upkeep" number={3}>

1. The next step is to register a new Upkeep using Chainlink Automation. Visit the Chainlink Automation App to get started. Click the button Register New Upkeep and connect your wallet.

1. You will be directed to a screen where you will choose your Trigger from a dropdown menu. Select Custom logic. Paste your address into the Target contract address field. Then, click Next.

1. You will be directed to a form to fill out more information on your upkeep. Fill out these details and set the gas limit to 200000 to ensure the Chainlink Automation network can perform the upkeep when the trigger conditions are met. Once all the information is filled in, click Register Upkeep. Wait until the upkeep has been registered and click View Upkeep. You'll be able to see all the details of your Upkeep.

</Accordion>

<Accordion title="Trigger the upkeep" number={4}>

1. Navigate back to the demo app page. Check off both boxes and click View Collection. Your screen will look similar to this:

```
<ClickToZoom
  src="/images/automation/qs-batch-reveal/batch-reveal7.png"
  alt="Batch Collection Quickstart Step 7"
  style="max-width: 70%;"
/>
```

1. Mint NFTs until the trigger condition is met. At that point, the demo app will display the message "Pending Batch Reveal". The Chainlink Automation upkeep will be triggered automatically. This will internally make a call to Chainlink VRF for randomness and assign the random traits to the tokens for the reveal.

1. If the reveal is a success, your screen will look similar to this. You have now completed the tutorial.

```
<ClickToZoom
  src="/images/automation/qs-batch-reveal/batch-reveal8.png"
  alt="Batch Collection Quickstart Step 8"
  style="max-width: 70%;"
/>
```

</Accordion>

chainlink-demo-app.mdx:

```
---
title: "Chainlink Demo App"
description: "Run the Chainlink Demo App locally to learn how to develop your own applications."
image: "QuickStarts-Chainlink-Demo-App.webp"
products: ["automation", "vrf", "feeds"]
time: "60 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside } from "@components"
```

Overview

The Chainlink Demo App is an end-to-end implementation of several Chainlink services. It uses the Hardhat development environment and the Next.js frontend framework. This guide shows you how to deploy this app yourself and use it as a starting point for building your own decentralized applications.

Try out the demo at chainlink-demo.app.

See the code on [GitHub](#).

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Objective

This guide shows you how to deploy a demo app, which requires several steps common to Web3 apps. You will learn the basics of how to use the Hardhat development environment and the Next.js frontend framework.

Before you begin

Before you start this tutorial, complete the following items:

- If you are new to smart contract development, learn how to Deploy Your First Smart Contract.
- Set up a cryptocurrency wallet such as MetaMask.
- To deploy this contract on testnets, ensure the deployer account has testnet ERC-677 LINK. Use the LINK faucet to retrieve testnet LINK.
- Install Node
- Install Yarn
- Install Git

Steps to implement

<Accordion title="Configure your local files" number={1}>

1. Clone the repo and change directories:

```
bash
git clone https://github.com/smartcontractkit/chainlink-fullstack && cd
chainlink-fullstack
```

1. Initialize the submodule:

```
bash
git submodule init
```

1. Update the submodule:

```
bash
git submodule update
```

1. Install the dependencies:

```
bash
yarn install
```

1. Start up the local Hardhat network and deploy all contracts:

```
bash
yarn chain
```

1. In a second terminal start up the local development server run the front-end app:

```
bash
yarn dev
```

To interact with the local network, follow this step-by-step guide on how to use MetaMask with a Hardhat node.

If you've set the mnemonic from MetaMask the first 20 accounts will be funded with ETH.

</Accordion>

<Accordion title="Configure environment variables" number={2}>

1. Use the .env.example file in the hardhat workspace. Copy it to new .env files and replace the values with the following values:

- NETWORKRPCURL - An RPC is required to deploy to public networks. You can obtain one from Infura.
- MNEMONIC - This is used to derive accounts from a wallet seed phrase from Metamask. The first account must have enough ETH to deploy the contracts as well as LINK which can be obtained from Chainlink's faucets.
- PRIVATEKEY - This is an alternative to using the mnemonic. Some changes are required in hardhat.config.js
- ETHERSCANAPIKEY - This can be obtained from an Etherscan account, and is used to verify your contract code on Etherscan.

1. In the frontend workspace, configure a new .env file with the following value:

- NEXTPUBLICINFURAKEY - Used for read-only mode and WalletConnect.

</Accordion>

<Accordion title="Deploy the contracts" number={3}>

1. Before deploying the RandomSVG contract on a public network, you need to create and fund a VRF subscription.

1. Set the ID of your VRF subscription in ./packages/hardhat/helper-hardhat-config.ts.

1. Deploy on a public network:

```
bash
yarn deploy --network sepolia
```

1. Confirm the contract onchain:

```
bash
npx hardhat verify --network <NETWORK> <CONTRACTADDRESS>
<CONSTRUCTORPARAMETERS>
```

example:

```
bash
```

```
npx hardhat verify --network sepolia
0x9279791897f112a41FfDa267ff7DbBC46b96c296
"0x9326BFA02ADD2366b30bacB125260Af641031331"
```

</Accordion>

<Accordion title="Run the example" number={4}>

Start the front end to test your deployed application.

1. Change to the frontend directory.

```
bash
yarn start
```

1. Navigate to the locally hosted web server to interact with the app.

When you are done, try making modifications to the code to improve it, redeploy the contract, and run the example again.

</Accordion>

Examine the code

You can learn how the frontend works with deployed contracts by analyzing the code on GitHub.

chainlink-hardhat-starter-kit.mdx:

```
---
title: "Chainlink Hardhat Starter Kit"
description: "Get started with Chainlink services inside your Hardhat projects."
image: "QuickStarts-Chainlink-Hardhat-Starter-Kit.webp"
products: ["general"]
time: "10 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Aside } from "@components"
```

Overview

Several Starter Kits are available for different smart contract development frameworks. These starter kits make it easy to get started with Chainlink services inside your Hardhat projects. This guide shows you how to install and configure the Chainlink Hardhat Starter Kit.

Objective

You will install and configure the Chainlink Hardhat starter kit so you can use it to integrate Chainlink services into your own smart contracts.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how

to interact with Chainlink's systems and services so that you can integrate them into your own. This template is

provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key

checks or error handling to make the usage of the product more clear. Do not

use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

- If you are new to smart contract development, learn how to Deploy Your First Smart Contract.
- If you are new to Hardhat, read the Hardhat Getting Started documentation.
- Set up a cryptocurrency wallet such as MetaMask.
- Install git
- Install Nodejs, and optionally install Yarn.

Set up your environment

1. Clone the repository and change directories.

```
shell
git clone https://github.com/smartcontractkit/hardhat-starter-kit/ && cd
hardhat-starter-kit
```

1. If you plan to use TypeScript, check out the TypeScript branch.

```
git checkout typescript
```

1. Install the dependencies using either npm or yarn:

```
npm install
```

```
yarn --install
```

1. Confirm that everything is configured correctly by running a test.

```
npx hardhat test
```

Using the Starter Kit

After you install the starter kit, you can complete several different tasks. See the Chainlink Hardhat Starter Kit Usage guide for a list of examples of how the starter kit can be used to develop smart contracts integrated with Chainlink services.

circuit-breaker.mdx:

```
---
title: "Data Feed Circuit Breaker Using Chainlink Automation"
description: "Build your own Data Feed Circuit Breaker Using Chainlink
Automation."
image: "QuickStarts-Circuit-Breaker.webp"
products: ["automation", "feeds"]
```

```
time: "90 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Address, Aside, ClickToZoom, CodeSample, CopyText } from
"@components"
```

Overview

Circuit breakers are useful for pausing dApps and processes when adverse events are detected onchain. These circuit breakers can prevent loss of assets during volatile market events, unstable network conditions, or if systems that your dApp relies on become compromised.

The example circuit breaker contract is a highly configurable proof of concept that can be used with Data Feeds. It is capable of emitting events or calling custom functions based on predefined conditions, and it comes with an interactive UI that allows users to easily configure and manage the contract.

```
{/ prettier-ignore /}
<div class="remix-callout">
  <a href="https://github.com/smartcontractkit/quickstarts-circuitbreaker">See
the code on GitHub</a>
</div>
```

```
<Aside type="caution" title="Disclaimer">
  This tutorial represents an example of using a Chainlink product or service
and is provided to help you understand how
  to interact with Chainlink's systems and services so that you can integrate
them into your own. This template is
  provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not
been audited, and may be missing key
  checks or error handling to make the usage of the product more clear. Do not
use the code in this example in a
  production environment without completing your own audits and application of
best practices. Neither Chainlink Labs,
  the Chainlink Foundation, nor Chainlink node operators are responsible for
unintended outputs that are generated due
  to errors in code.
</Aside>
```

Objective

The Circuit Breaker repository contains an example implementation of a circuit breaker that can be used with any OCR price feed. You can monitor a given contract, specify the price when the circuit breaker will be triggered based on predefined conditions, and specify the underlying logic of what happens when the circuit breaker is triggered.

Before you begin

```
<Aside type="note" title="New to smart contracts?">
  This tutorial assumes that you know how to create and deploy basic smart
contracts. If you are new to smart contract
  development, deploy a VRF-compatible contract by following these
instructions and then return to this tutorial.
</Aside>
```

You can run this example on Linux, MacOS, or the Windows Subsystem for Linux.

Before you start this tutorial, install the required tools:

- Install git
- Run `git --version` to check the installation. You should see an output

similar to git version x.x.x.

- Install Nodejs 16.0.0 or higher
- Run `node --version` to check the installation. You should see an output similar to v16.x.x.

- Install and configure a cryptocurrency wallet like MetaMask.

- Add the Avalanche Fuji testnet and LINK to your wallet:

```
{ " " }
<a
  class="erc-token-address"
  id="431130x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846"
  href="/resources/link-token-contracts#fuji-testnet"
>
  Avalanche Fuji testnet
</a>
```

- Add testnet funds to your wallet. You will need both ERC-677 LINK and the gas currency for the chain where your contract is deployed. For this example, use Avalanche Fuji and testnet LINK. You can get both at faucets.chain.link.

Steps to implement

<Accordion title="Setup the example" number={1}>

To set up this example, clone the source and install the required packages.

1. Clone the smartcontractkit/quickstarts-circuitbreaker repository and change directories:

```
shell
git clone https://github.com/smartcontractkit/quickstarts-circuitbreaker.git
&& \
cd quickstarts-circuitbreaker
```

1. Install the chainlink/contracts npm package:

```
shell
npm install @chainlink/contracts@0.6.1 --save
```

For this tutorial, disregard the resulting npm warnings.

</Accordion>

<Accordion title="Deploy contracts" number={2}>

1. Open the ExampleImplementation.sol contract in Remix:

```
{/ prettier-ignore /}
<div class="remix-callout">
  <a
href="https://remix.ethereum.org?url=https://github.com/smartcontractkit/
quickstarts-circuitbreaker/blob/main/ExampleImplementation.sol">Open in
Remix</a>
</div>
```

1. On the Solidity compiler tab, click Compile ExampleImplementation.sol.

1. On the Deploy and run transactions tab, select Injected Provider - MetaMask for the Environment field. Make sure the ExampleImplementation.sol contract is selected in the Contract field.

1. Scroll down to the Deploy section:

```
{ " " }
```

<ClickToZoom src="/images/quickstarts/circuit-breaker/example-implementation-

deploy.png" />

Click the dropdown carat to expand the Deploy section:

{" "}

<ClickToZoom src="/images/quickstarts/circuit-breaker/example-implementation-deploy-expanded.png" />

1. Enter the following input to the constructor:

Item	Value
MAXBALANCE	<CopyText text="20000000000000" code/>
FEEDADDRESS	<Address contractUrl="https://testnet.snowtrace.io/address/0x31CF013A08c6Ac228C94551d535d5BAfE19c602a" />
EMERGENCYPOSSIBLE	<CopyText text="true" code/>

- The maxBalance in sats. For example, <CopyText text="20000000000000" code/> for a max balance of \$20,000.
- The proxy contract address of the price feed. For example, the address for the BTC/USD price feed on Fuji is <Address contractUrl="https://testnet.snowtrace.io/address/0x31CF013A08c6Ac228C94551d535d5BAfE19c602a" />. Find other price feed addresses here.
- The isEmergencyPossible flag: set this to true if you want the circuit breaking condition to be checked. You can turn off this check by setting this value to false.

1. Click Deploy and confirm the transaction in your wallet.

Next, deploy the CircuitBreaker.sol contract:

1. Open the CircuitBreaker.sol contract in Remix:

```
{/ prettier-ignore /}  
<div class="remix-callout">  
  <a  
href="https://remix.ethereum.org?url=https://github.com/smartcontractkit/  
quickstarts-circuitbreaker/blob/main/CircuitBreaker.sol">Open in Remix</a>  
</div>
```

1. On the Solidity compiler tab, click Compile CircuitBreaker.sol.
1. On the Deploy and run transactions tab, select Injected Provider - MetaMask for the Environment field. Make sure the CircuitBreaker.sol contract is selected in the Contract field.
1. There are no constructor inputs for this contract. Click Deploy and confirm the transaction in your wallet.

</Accordion>

<Accordion title="Register and fund an upkeep" number={3}>

Registering an upkeep on Chainlink Automation creates a smart contract that will run your circuit breaker contract.

```
{/ prettier-ignore /}  
{" "}
```



```
<div class="remix-callout">
  <a href="https://automation.chain.link">Open the Chainlink Automation App</a>
</div>
```

1. Click Register new Upkeep.

1. Select the Custom logic trigger.

1. For Target contract address, input the address of your deployed CircuitBreaker.sol contract. You can find this in your wallet's transaction history or copy it from Remix:

```
{" "}
```

```
<ClickToZoom src="/images/quickstarts/circuit-breaker/circuit-breaker-copy-address.png" />
```

There may be a warning that says "Unable to verify if this is an Automation compatible contract." The CircuitBreaker.sol contract does implement the AutomationCompatibleInterface so you can disregard this warning. Click Next.

1. For Upkeep details, add the following:

- A name for your upkeep
- A starting balance of 2 testnet LINK
- The admin address and gas limit are prefilled for you.
- Although the Check data field is marked optional, it is required for this tutorial. You'll fill this in during the next step.

1. For your upkeep's Check data field, you need to ABI encode the address of your deployed ExampleImplementation.sol contract.

1. Navigate to the HashEx Online ABI Encoder.

1. Click Add argument and select Address as the argument type from the dropdown menu. 1. Enter the address of your deployed ExampleImplementation.sol contract.

```
<ClickToZoom src="/images/quickstarts/circuit-breaker/hashex-abi-encoder-address.png" />
```

1. From the Encoded data field, copy the encoded address.

1. Navigate back to the Chainlink Automation app. In the Check Data field, enter 0x and paste in the ABI encoded address of the ExampleImplementation.sol contract.

1. Click Register upkeep and confirm the transaction in your wallet.

```
</Accordion>
```

```
<Accordion title=" Check emergency action" number={4}>
```

Now the Chainlink Automation network will watch your contract for these trigger parameters. If the price from the feed you provided is above the maxBalance threshold that was specified, the executeEmergencyAction() function will trigger. As defined in ExampleImplementation.sol, the function increments a counter:

```
solidity
/
  executes emergency action
/
function executeEmergencyAction() external {
  counter += 1;
  circuitbrokenflag = true;
```

```
    emit emergencyActionPerformed(counter, feedAddress);
}
```

1. Navigate to your upkeep in the Chainlink Automation app. If the `executeEmergencyAction()` function is triggered, you will see "Perform upkeep" logs listed in the History section.

1. Navigate to Remix and expand the details for your deployed `ExampleImplementation.sol` contract. Click the counter button to view the value of the counter variable.

1. If the counter was incremented once, you should see `0: uint256: 1` as the output:

```
<ClickToZoom src="/images/quickstarts/circuit-breaker/check-counter.png" />
```

</Accordion>

```
<Accordion title="Update the example implementation contract" number={5}>
```

If you want to update your `ExampleImplementation.sol` contract, you can redeploy it with updated values and then use the same Automation upkeep. For example, to update the `maxBalance`:

1. Redeploy the `ExampleImplementation.sol` contract with your updated `maxBalance` value.

1. ABI encode the address of your newly deployed `ExampleImplementation.sol` contract.

1. In the Chainlink Automation app, within the Actions menu for your existing upkeep, click Edit check data:

```
<ClickToZoom src="/images/quickstarts/circuit-breaker/automation-edit-checkdata.png" />
```

1. Enter `0x`, paste the new ABI-encoded contract address, and click Change check data. MetaMask opens and prompts you to confirm the transaction.

```
<ClickToZoom src="/images/quickstarts/circuit-breaker/automation-change-checkdata.png" />
```

You can use the same process to make other changes to your example implementation contract, like changing the logic in `executeEmergencyAction()` or changing the price feed that you're monitoring.

</Accordion>

Cleanup

If you want to experiment further with this tutorial, you can pause the Automation upkeep so that it stops running while you work on updating your example implementation contract. Otherwise, you can cancel the upkeep to reclaim any unused testnet LINK back to your wallet. Both options are located in the Chainlink Automation app within the Actions menu on your upkeep's details page.

```
# deploy-your-first-contract.mdx:
```

```
---
```

```
title: "Deploy Your First Smart Contract"
```

```
description: "Write your first smart contract and run it in your browser without any knowledge about Ethereum or blockchains."
```

```
image: "QuickStarts-Deploy-your-first-Smart-Contract.webp"
```

```
products: ["general"]
```

```
time: "10 minutes"
```

```
requires: "Wallet with gas token & ERC-677 LINK"
```

```
---
```

```
import { Accordion, Aside, CodeSample } from "@components"
```

Overview

You can write your first smart contract and run it in your browser without any knowledge about Ethereum or blockchains. This guide shows you how easy it is to develop smart contracts using the Solidity language, a MetaMask wallet and the Remix Development Environment. You can use all of these tools in your browser for free with no signup required.

Objective

You will create and deploy a simple "Hello world" smart contract using the following process:

1. Write: Write a contract to define how the contract functions, what data it can store, what other contracts it interacts with, and what external APIs it might call.

1. Compile: Pass your smart contract code through a compiler to translate the contract into byte code that the blockchain can understand. For example, Solidity code must be compiled before it can run in the Ethereum Virtual Machine.

1. Deploy: Send the compiled smart contract to the blockchain. From that point forward, the contract cannot be altered. However, you can still interact with the contract in several ways.

1. Run functions: When you run the functions that you defined for the contract, the network processes those functions and modifies the state of your contract. For some functions, the network charges a small fee to complete the work. Your contract can also have functions that transfer funds to other contracts or wallets.

This guide walks you through each step.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how

to interact with Chainlink's systems and services so that you can integrate them into your own. This template is

provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key

checks or error handling to make the usage of the product more clear. Do not use the code in this example in a

production environment without completing your own audits and application of best practices. Neither Chainlink Labs,

the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due

to errors in code.

</Aside>

Steps to implement

<Accordion title="Install and fund your MetaMask wallet" number={1}>

Deploying smart contracts onchain requires a wallet and ETH. The ETH pays for the work required by the Ethereum network to add the contract to the blockchain and store the variables. The wallet holds the ETH that you need to pay for the transaction. Install MetaMask, configure it to use the Sepolia testnet, and fund your wallet with free testnet ETH.

1. Install and configure the MetaMask extension in your browser.

1. After you install the extension, open your browser extension list and click MetaMask to open MetaMask.

!Screenshot showing the browser extension list with MetaMask installed.

1. Follow the instructions in MetaMask to create a new MetaMask wallet. The new wallet includes a 12-word mnemonic phrase. This phrase is the key to your wallet. Copy that phrase down in a very secure location that only you can access. You can use this phrase to retrieve your wallet later or add it to another browser.

1. Set MetaMask to use the Sepolia test network.

!Screenshot showing the network selection menu in MetaMask. The Sepolia Test Network is selected.

1. Go to faucets.chain.link and follow the steps to send testnet ETH to your MetaMask wallet address. After the faucet completes the transaction, you should have testnet ETH in your MetaMask wallet on the Sepolia testnet. Optionally, you can get testnet ETH from one of the alternative faucets.

Now that you configured your wallet and funded it with testnet ETH, you can write, compile, and deploy your contract.

</Accordion>

<Accordion title="Write, compile, and deploy your first smart contract" number={2}>

Your first contract is a simple HelloWorld.sol example. This example shows you how to set and retrieve variables in a smart contract onchain.

<CodeSample src="samples/Tutorials/HelloWorld.sol" />

1. Open the example contract in the Remix IDE. Remix opens and shows the contents of the smart contract. You can modify the code in this editor when you write your own contract.

```
{/ prettier-ignore /}
```

```
<CodeSample src="samples/Tutorials/HelloWorld.sol" showButtonOnly/>
```

1. Because the code is already written, you can start the compile step. On the left side of Remix, click the Solidity Compiler tab to view the compiler settings.

!Screenshot showing the Compiler tab and its settings.

1. For this contract, use the default compiler settings. Click the Compile HelloWorld.sol button to compile the contract. This converts the contract from Solidity into bytecode that the Ethereum Virtual Machine can understand. Remix automatically detects the correct compiler version depending on the pragma that you specify in the contract.

!Screenshot of the Compile button.

1. After Remix compiles the contract, deploy it. On the left side of Remix, click the Deploy and Run tab to view the deployment settings.

!Screenshot of the Deploy tab and its settings.

1. In the deployment settings, select the Injected Provider environment. This tells Remix that you want to deploy your contract to the blockchain that you configured in MetaMask. You could optionally use one of the Javascript VM options, but they run in a virtual environment with no connection to an actual

blockchain or Chainlink oracles.

!Screenshot showing the Injected Provider environment selected.

1. Next to the Deploy button, enter a message that you want to send with the smart contract when you deploy it. This contract has a constructor that sets an initial message when you deploy the contract.

!Screenshot of the Deploy button with "Hello world!" as the defined message.

1. Click the Deploy button to deploy the contract and its initial message to the blockchain network. MetaMask opens and asks you to confirm payment to deploy the contract. Make sure MetaMask is set to the Sepolia network before you accept the transaction. Because these transactions are on the blockchain, they are not reversible.

1. In the MetaMask prompt, click Confirm to approve the transaction and spend your testnet ETH required to deploy the contract.

!Screenshot showing MetaMask asking you to confirm the transaction.

1. After a few seconds, the transaction completes and your contract appears under the Deployed Contracts list in Remix. Click the contract dropdown to view its variables and functions.

!Screenshot showing the deployed Hello World contract.

1. Click the message variable. Remix retrieves and prints the initial message that you set.

!Screenshot showing the message function and the returned "Hello World" message.

The contract has an address just like your wallet address. If you save this address, you can return to your deployed contract at any time to retrieve variables or execute functions. To see details about your deployed contract, copy the contract address from the list in Remix and search for it in the Etherscan Sepolia Testnet Explorer.

</Accordion>

<Accordion title="Run functions in your contract" number={3}>

Because you deployed the contract to an actual blockchain, several nodes on the test network confirmed your payment for the smart contract. The contract, its variables, and its functions remain in the blockchain permanently. To change the message variable that is stored with your contract, run the updateMessage function.

1. In your deployed contract, enter a new message next to the updateMessage function.

!Screenshot showing the updateMessage function with a new value.

1. Click the updateMessage button to set the new message in the contract data. MetaMask opens and asks you to confirm payment to update the state of your contract.

1. In the new MetaMask prompt, click Confirm to approve the transaction.

!Screenshot showing MetaMask asking you to confirm the transaction.

1. Click the message variable again to see the updated value. It might take a few seconds before the transaction updates the variable.

!Screenshot showing the updated value for the message value.

</Accordion>

Now you know how to deploy example contracts to a test network and run the functions in those contracts. You can write your own contracts and test them using this same process.

Next, read the Consuming Data Feeds guide to learn how to connect your smart contracts to Chainlink Data Feeds and retrieve onchain data that your smart contracts can act on.

dev3-chainlink-sdk.mdx:

```
---
title: "Dev3 Chainlink SDK"
description: "Learn how to install and use the Dev3 Chainlink SDK in your applications."
image: "QuickStarts-Dev3-Chainlink-SDK.webp"
products: ["general"]
time: "30 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Aside } from "@components"
import { TabsContent } from "@components/Tabs"
```

Overview

The Dev3 Chainlink SDK module is a fully open source TypeScript SDK which enables any frontend developer to fetch the prices of various assets through Chainlink Data Feeds. The SDK works with vanilla npm projects and frontend frameworks like Angular and React. Users can fetch price pairs and other data.

Objective

You will install and use the Dev3 SDK to monitor a Chainlink price feed. Optionally, you can also deploy a Dev3 Chainlink SDK template from the Dev3 dashboard.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

Before you start this tutorial, complete the following items:

- Install git
- Run git --version to check the installation. You should see an output

similar to git version x.x.x.

- Install Nodejs 16.0.0 or higher

- Run `node --version` to check the installation. You should see an output similar to v16.x.x.

Install and initialize the SDK

Install the Dev3 SDK in your npm project.

1. Type the following command on your terminal while on your project's root folder:

```
sh
npm install dev3-sdk
```

1. In your JavaScript or TypeScript file:

1. Import the Chainlink module:

```
ts
import { Chainlink } from "dev3-sdk"
```

1. Get an RPC URL for the network where you want to monitor price feeds. For example, a public node for Ethereum is <https://ethereum.publicnode.com>.

1. Fill in your RPC URL and then initialize the SDK by calling:

```
ts
const ethSDK = Chainlink.instance("https://ethereum.publicnode.com",
Chainlink.PriceFeeds.ETH)
```

This sets up reading data feeds for Ethereum mainnet.

Consuming data feeds

Once the SDK is initialized, you can start consuming different feeds as outlined in the examples below:

```
{/ prettier-ignore /}
```

```
<TabContent sharedStore="feedType" client:visible>
```

```
  <Fragment slot="tab.1">Price Feeds</Fragment>
```

```
  <Fragment slot="tab.2">Proof of Reserve Feeds</Fragment>
```

```
  <Fragment slot="panel.1">
```

1. Add the following code to your JavaScript or TypeScript file:

```
ts
// AAVE/ETH price feed
ethSDK.getFromOracle(ethSDK.feeds.AAVEETH).then((res) => {
  console.log(res.answer.toString());
});
```

1. Run your file. For example:

```
sh
node index.js
```

1. The output should be similar to: 401041900000000000. Compare this to the AAVE / ETH price feed, which returns $\hat{0.04010419}$ at the time of writing.

```
</Fragment>
```

```
<Fragment slot="panel.2">
```

1. Add the following code to your JavaScript or TypeScript file:

```
ts
// HBTC PoR
```

```
ethSDK.getFromOracle(ethSDK.feeds.HBTCPOr).then((res) => {
  console.log(res.answer.toString());
});
```

1. Run your file. For example:

```
sh
node index.js
```

1. The output should be similar to: 9708077983300000000000. Compare this to the HBTC PoR price feed, which returns 970.81 at the time of writing.

</Fragment>

</TabsContent>

Full list of feeds

You can find all available feeds by visiting [data.chainlink](#).

Reading data

The data is returned in the form of a RoundDataModel object:

```
ts
export interface RoundDataModel {
  roundID: BigNumber
  answer: BigNumber
  formattedAnswer?: string
  startedAt: BigNumber
  updatedAt: BigNumber
  answeredInRound: BigNumber
  assetName?: string
  dataFeedName?: string
}
```

Deployable templates

The Dev3 SDK includes Chainlink interfaces and contracts that you can deploy from the Dev3 dashboard. You can use these interfaces and contracts from within your frontends and/or wallets.

To deploy them, navigate to the Dev3 dashboard and choose the Deploy from template option in your workspace. Search for "chainlink" within your Dev3 workspace, or view the source code directly:

- Interfaces
- Deployable contracts

Additional information

- The source code for this extension is extracted to the GitHub repo so you can also use it as a standalone module.
- Dev3 Chainlink SDK extracts all the pair contracts addresses for all networks that are compatible into code generated classes, so all modern editors will support full code autocomplete.

!Contract address pairs screenshot

dynamic-metadata.mdx:

title: "Dynamic Metadata with Automation"

description: "Use Chainlink Automation to create NFTs that can change over time, based on user interaction, or any number of conditions."


```
image: "QuickStarts-Dynamic-Metadata-with-Automation.webp"
products: ["automation"]
time: "60 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside, ClickToZoom } from "@components"
import { YouTube } from "@astro-community/astro-embed-youtube"
```

Video

Learn how to create dynamic NFTs by following along with this video tutorial:

```
<YouTube id="https://www.youtube.com/watch?v=E7Rm1LUKhj4" />
```

```
{/ prettier-ignore /}
<div class="remix-callout">
  <a href="https://github.com/smartcontractkit/smart-contract-examples/tree/main/dynamic-nft">See the code on GitHub</a>
</div>
```

Objective

In this tutorial, you will use the Chainlink Automation Network to help create a dynamic NFT by using the Automation-compatible smart contract provided in the repository.

The demo includes:

- Setting up and funding an upkeep on the Chainlink Automation Network
- Setting up a dynamic NFT and viewing the progression of the NFT on OpenSea.

You can use this example as a starting point for building your own dynamic NFTs and provide more interactive experiences with your collections.

Before you begin

Before you start this tutorial, complete the following items:

- If you are new to smart contract development, learn how to Deploy Your First Smart Contract.
- Set up a cryptocurrency wallet
- To deploy this contract on testnets, ensure the deployer account has testnet LINK. Use the LINK faucet to retrieve testnet LINK.
- Open a new Remix project.

Steps to implement

```
<Accordion title="Compile and deploy your contract" number={1}>
```

1. Open the 2complete.sol contract in Remix and rename the contract to dynNFT.sol.

```
  {/ prettier-ignore /}
  <div class="remix-callout">
    <a
href="https://remix.ethereum.org/#url=https://github.com/smartcontractkit/smart-
contract-examples/blob/main/dynamic-nft/2complete.sol">Open in Remix</a>
  </div>
```

1. Compile the contract in Remix. Under the section titles DEPLOY & RUN TRANSACTIONS you will have to change some values in the dropdown menus. Under ENVIRONMENT, select Injected Provider - MetaMask or WalletConnect depending on the cryptocurrency wallet you are using. Selecting one of these options will

prompt you to connect your wallet with Remix.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-1.png"
  alt="Dynamic Metadata Quickstart Step 2"
  style="max-width: 70%;"
/>
```

1. Ensure that the CONTRACT field displays dynNFT. Finally, for the Deploy value, you must enter a time interval value denoting when the dynamic NFT will change. For this demo, we have inputted 30, but you can change this value. Once you have entered your interval, click the orange Deploy button and follow the prompts to allow the transaction to be performed. This may take a few minutes depending on factors such as network traffic.

</Accordion>

<Accordion title="Mint an NFT" number={2}>

1. Once the contract is deployed, you should be able to see more information under the Deployed Contracts section.

At this point, you can mint an NFT. In the field next to safeMint paste your wallet address. Then, click the orange transact button and follow the prompts to confirm the transaction.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-2.png"
  alt="Dynamic Metadata Quickstart Step 4"
  style="max-width: 70%;"
/>
```

1. After minting the NFT, you can look at the stage of the NFT based on the tokenURIs given in the contract. Enter "0" in the flowerStage field and click the blue call button. You should see the stage of the NFT.

Enter 0 in the tokenURI field and click the blue call button to see the corresponding URI for the NFT.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-3.png"
  alt="Dynamic Metadata Quickstart Step 5.1"
  style="max-width: 70%;"
/>
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-4.png"
  alt="Dynamic Metadata Quickstart Step 5.2"
  style="max-width: 70%;"
/>
```

</Accordion>

<Accordion title="Register an upkeep" number={3}>

1. Copy the contract address and open the Chainlink Automation Interface. Connect your wallet to the network and click the blue button Register New Upkeep. For the Trigger mechanism, select Custom logic. You will be prompted to enter your contract address. Paste your address and click Next.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-5.png"
  alt="Dynamic Metadata Quickstart Step 6"
  style="max-width: 70%;"
/>
```

1. You must fill out the prompts to finish registering upkeep. For the Starting balance (LINK), enter 0.5 for this demo. After entering information, click Register Upkeep. It may take a few moments for this to complete.

</Accordion>

<Accordion title="Trigger the upkeep" number={4}>

1. Navigate back to Remix. Under the Deployed Contracts section, enter [] in the checkUpkeep field and click the blue button. You may see the following text: upkeepNeeded true. This means that an upkeep will be performed.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-10.png"
  alt="Dynamic Metadata Quickstart Step 8"
  style="max-width: 70%;"
/>
```

1. While waiting for the upkeep to be performed, copy the contract address to your clipboard and open OpenSea (<https://testnets.opensea.io/>). Paste your address in search bar at the top of the page. You should be directed to a page with the image of your minted NFT. Click the seed thumbnail to learn more about the NFT.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-6.png"
  alt="Dynamic Metadata Quickstart Step 9"
  style="max-width: 70%;"
/>
```

1. Navigate back to Remix. Enter [] in the checkUpkeep field and click the blue button. If this returns upkeepNeeded false your upkeep should have been performed. Navigate to the Chainlink Automation Interface and select your upkeep to view more details and to confirm that an upkeep was performed. If upkeepNeeded false is not returned in Remix, you may have to wait a few minutes and run the checkUpkeep function.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-7.png"
  alt="Dynamic Metadata Quickstart Step 10"
  style="max-width: 70%;"
/>
```

</Accordion>

<Accordion title="Check the NFT stage" number={5}>

1. Once the upkeep has been performed, navigate to Remix and click flowerStage and tokenURI to see the most recent flower stage and corresponding URI. Navigate back to the information of your NFT on OpenSea. Click the button on the top right to refresh the metadata. You should see the NFT become a bloom.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-8.png"
  alt="Dynamic Metadata Quickstart Step 11"
  style="max-width: 70%;"
/>
```

1. To see the NFT become a fully blossomed flower, you will have to wait another 30 seconds (or the amount of time you denoted when deploying your contract). After this interval, refresh the NFT's metadata again. You should see a fully blossomed flower. You have now completed the tutorial.

```
<ClickToZoom
  src="/images/automation/qs-dynnft/dynnft-qs-9.png"
  alt="Dynamic Metadata Quickstart Step 12"
  style="max-width: 70%;"
/>
```

</Accordion>

Summary

This tutorial used Chainlink Automation to create a dynamic NFT. This was done by deploying an Automation compatible contract in Remix and registering an Upkeep with the Chainlink Automation network. The NFT changed every 30 seconds based on the time interval we inputted when deploying the contract. The final NFT stage was a fully bloomed flower.

```
# eth-balance-monitor.mdx:
```

```
---
title: "Automate Contract Balance Top-up"
description: "Automate the process of maintaining an Ethereum balance in your smart contract using Chainlink Automation."
image: "QuickStarts-ETH-Balance-Monitor.webp"
products: ["automation"]
time: "30 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside, CodeSample } from "@components"
```

Objective

Some smart contract applications require you to maintain a certain Ethereum balance to ensure optimal operation. You can automate this process using Chainlink Automation. Use these smart contracts to monitor and top-up native token balances on your smart contracts.

This Automation contract monitors and funds Ethereum addresses that developers might need to top up frequently based on a configurable threshold. You will deploy this example monitor contract and use the `automation.chain.link` interface to register an upkeep and run the contract. To learn the basics about Chainlink Automation, read the [Chainlink Automation Overview](#).

For this example, use the Ethereum Sepolia testnet to simplify access to testnet funds. You will top up an address with a specific amount of testnet ETH. Later, you can use another EVM-compatible network of your choice.

Before you begin

Before you start this tutorial, complete the following items:

- If you are new to smart contract development, learn how to [Deploy Your First Smart Contract](#).
- Set up a cryptocurrency wallet such as MetaMask.
- Fund your wallet with the following testnet tokens:
 - Get testnet ETH from [faucet.polygon.technology](#) to pay for your onchain transactions.
 - Get ERC-677 testnet LINK from [faucets.chain.link/fuji](#).

Steps to implement

```
<Accordion title="Deploy the upkeep contract" number={1}>
```

1. Open the EthBalanceMonitor.sol contract in Remix.

```
{/ prettier-ignore /}  
<CodeSample src="samples/Automation/tutorials/EthBalanceMonitor.sol"  
showButtonOnly />
```

1. Compile the contract in Remix.

1. On the Deploy & Run Transactions tab in Remix, set your Environment to Injected Provider and allow Remix to connect to MetaMask.

1. The constructor requires a minWaitPeriodSeconds value. Specify this in the Deploy fields in Remix. For this example using Ethereum Sepolia, enter the following value:

- minWaitPeriodSeconds: 60

1. Click Transact to deploy the contract with the defined constructor settings and accept the transaction in MetaMask. After the contract is deployed, it will appear in your Deployed Contracts list with several functions available to configure.

Note: You will come back and set your forwarder address later. This forwarder address becomes available after you create your upkeep.

1. Use MetaMask to Send 0.001 ETH to your contract address. The funds are used to top-up other contracts with ETH. See the Fund Your Contracts page to learn how to find your contract address and send funds to those contracts. For this example, 0.001 ETH is sufficient for demonstration purposes. In a production environment, you might store a larger amount of funds in the contract so Chainlink Automation can automatically distribute them to several different addresses as they are needed over time.

</Accordion>

<Accordion title="Configure the address watch list" number={2}>

Before your contract will fund an address, you must set the watchList address array with minBalancesWei and topUpAmountsWei variables. For demonstration purposes, you configure your own wallet as the top-up address. This makes it easy to see the ETH being sent as part of the automated top-up function. After you complete this tutorial, you can configure any wallet or contract address that you want to keep funded.

1. In the list of functions for your deployed contract, run the setWatchList function. This function requires an addresses array, a minBalancesWei array that maps to the addresses, and a topUpAmountsWei array that also maps to the addresses. In Remix, arrays require brackets and quotes around both addresses and integer values. For this example, set the following values:

- addresses: ["YOURWALLETADDRESSORCONTRACTADDRESS"]
- minBalancesWei: ["1000000000000000000000"]
- topUpAmountsWei: ["00010000000000000000"]

These values tell the top up contract to top up the specified address with 0.0001 ETH if the address balance is less than 100 ETH. These settings are intended to demonstrate the example using testnet faucet funds. For a production application, you might set more reasonable values that top up a smart contract with 10 ETH if the balance is less than 1 ETH.

1. After you configure the function values, click transact to run the function. MetaMask asks you to confirm the transaction.

1. In the functions list, click the getWatchList function to confirm your

settings are correct.

</Accordion>

<Accordion title="Register the upkeep" number={3}>

Now that the contract is deployed, funded, and configured, register the upkeep and Chainlink Automation will automatically run the example code.

1. Go to the `automation.chain.link` registration UI.

1. Click Connect wallet in the top right of the UI to connect your wallet.

1. After the site recognizes your wallet, click Register new Upkeep. The registration interface opens.

1. Select the Custom logic trigger option.

1. Copy your contract address from Remix and paste it into the Target contract address field.

1. Click Next to go to the Upkeep details page.

1. Specify a name for your upkeep and set a Starting balance of 2 LINK. Leave the other settings at their default values.

1. Click Register Upkeep to complete the registration process.

1. Go to the Upkeep details page and copy your Forwarder address.

1. Navigate back to Remix and find the `setForwarderAddress` function. Paste your forwarder address and click click transact to run the function. MetaMask asks you to confirm the transaction.

Now that you've registered the upkeep and configured your contract with your new upkeep's forwarder address, Chainlink Automation handles the rest of the process.

</Accordion>

<Accordion title="Check your upkeeps" number={4}>

The upkeep runs the `checkUpkeep` function in your contract, which checks the balances of all addresses in the `watchList` against their defined limits. If any of the addresses are below their specified minimum balances, the upkeep runs the `topUp` function for that specific address, which distributes the ETH in your contract to that address. Unless your wallet happens to have more than 100 testnet ETH, you will receive ETH in your wallet every 60 seconds as defined by `minWaitPeriodSeconds` in your contract. Check to make sure the example is running correctly:

1. Go to `automation.chain.link`, view your new upkeep, and confirm that it is performing upkeeps in the History list.

1. View your contract address and your wallet address at the Sepolia block explorer to see the transactions happening between your contract and your wallet address.

The example continues to run until your upkeep runs out of LINK, your contract runs out of ETH, or until you manually pause or cancel the upkeep.

After you are done with this example, you can run the `withdraw` function on your contract to withdraw any remaining testnet ETH so you can use it later. Also, you can cancel your upkeep in the Automation UI and withdraw any remaining

unspent LINK.

</Accordion>

Examine the example code

The EthBalanceMonitor contract is ownable, pauseable, and compatible with the AutomationCompatibleInterface contract:

- Ownable: The contract has an owner address, and provides basic authorization control functions. This simplifies the implementation of user permissions and allows for transfer of ownership.
- Pauseable: This feature allows the contract to implement a pause and unpause mechanism that the contract owner can trigger.
- Compatible: The AutomationCompatibleInterface is necessary to create contracts that are compatible with the Chainlink Automation Network. To learn more about the AutomationCompatibleInterface and its uses and functions, read the Making Compatible Contracts guide.

<Aside type="note" title="Note on Owner Settings">

Aside from certain features listed below, only owners can withdraw funds and pause or unpause the contract. If the

contract is paused or unpaused, it will affect checkUpkeep, performUpkeep, and topUp functions.

</Aside>

Contract functions

Functions with an asterisk (*) denote features that only the owner can change. Click on each function to learn more about its parameters and design patterns:

Function Name	Description
-----	-----
setWatchList\	Addresses to watch minimum balance and how much to top it up.
setForwarderAddress\	Updates your upkeep's forwarder address, which is available after you create your upkeep.
setMinWaitPeriodSeconds\	Updates the global minimum period between top ups.
topUp	Used by performUpkeep. This function will only trigger top up if conditions are met.

Below are the feed functions in EthBalanceMonitor:

Read Function Name	Description
-----	-----
getUnderfundedAddresses	View function used in checkUpkeep to find underfunded balances.
getForwarderAddress	Views the upkeep's forwarder address.
getMinWaitPeriodSeconds	Views the global minimum period between top ups.
getWatchList	Views addresses to watch minimum balance and how much to top it up.
getAccountInfo	Provides information about the specific target address, including the last time it was topped up. This function is external

only. |

setWatchList Function

Parameters

Name	Description	Suggested Setting
-----	-----	
addresses	The list of addresses to watch	(not applicable)
minBalancesWei	The minimum balances for each address	50000000000000000000
(5 ETH)		
topUpAmountsWei	The amount to top up each address	50000000000000000000
(5 ETH)		

Only the owner can setWatchList. Each of the parameters should be set with distinct requirements for each address.

setForwarderAddress Function

Parameters

Name	Description
-----	-----
forwarderAddress	Unique forwarder address used to secure your upkeep

Only the forwarderAddress can performUpkeep. Your upkeep's forwarder address can be found on the Chainlink Automation app after you create the upkeep. Only the owner can set a new forwarderAddress after deployment.

setMinWaitPeriodSeconds Function

Parameters

Name	Description	Suggested Setting
-----	-----	
period	Minimum wait period (in seconds) for addresses between funding	3600
(1 hour)		

period denotes the length of time between top ups for a specific address. This is a global setting that prevents draining of funds from the contract if the private key for an address is compromised or if a gas spike occurs. However, only the owner can set a different minimum wait period after deployment.

topUp Function

Parameters

Name	Description
-----	-----
needsFunding	List of addresses to fund (addresses must be pre-approved)

Any address can trigger the topUp function. This is an intentional design pattern that shows how easy it is to make an existing contract compatible with Chainlink Automation while maintaining an open interface. All validations are performed before the funding is triggered. If the conditions are not met, any attempt to top up will revert.

foundry-chainlink-toolkit.mdx:


```
---
title: "Foundry Chainlink Toolkit"
description: "Test smart contracts and local Chainlink nodes using Foundry."
image: "QuickStarts-Foundry-Chainlink-Toolkit.webp"
products: ["general"]
time: "10 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside } from "@components"
```

Overview

This toolkit makes spinning up, managing, and testing smart contracts and local Chainlink nodes easier using Foundry to deploy and test smart contracts. It can be integrated into existing Foundry projects.

Objective

This project shows you how to install and run the simplify the development and testing of smart contracts that use Chainlink oracles. This project is aimed primarily at those who use the Foundry toolchain.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided as IS and as AVAILABLE without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

1. Install Foundry toolchain. Reference the below commands or go to the Foundry documentation.

- MacOS/Linux

```
curl -L https://foundry.paradigm.xyz | bash
```

This will download foundryup. Restart your terminal session, then install Foundry by running:

```
foundryup
```

You might see the following error on MacOS: dyld: Library not loaded: /usr/local/opt/libusb/lib/libusb-1.0.0.dylib To fix this, install libusb: brew install libusb See installation troubleshooting for details.

1. Install GNU make. The functionality of the project is wrapped in the makefile. Reference the below commands based on your OS or go to Make

documentation.

- MacOS: install Homebrew first, then run

```
brew install make
```

- Debian/Ubuntu

```
apt install make
```

- Fedora/RHEL

```
yum install make
```

1. Install and run Docker.

1. Integrate the toolkit into your project. The Foundry-Chainlink toolkit is designed to be installed as a Forge dependency:

```
forge install smartcontractkit/foundry-chainlink-toolkit
```

1. Install the Forge Standard Library in your project:

```
forge install foundry-rs/forge-std
```

1. If you need to run this toolkit as a demo standalone application, install the dependencies:

```
git submodule update
```

Steps to implement

<Accordion title="Set up environment variables" number={1}>

Use the env.template file, create or update an .env file in the root directory of your project.
In most cases, you will not need to modify the default values specified in this file.

The following environment variables are available:

- FCTPLUGINPATH - path to the Foundry-Chainlink toolkit root
- ETHURL - RPC node web socket used by the Chainlink node
- RPCURL - RPC node http endpoint used by Forge
- PRIVATEKEY - private key of an account used for deployment and interaction with smart contracts. Once Anvil is started, a set of private keys for local usage is provided. Use one of these for local development
- ROOT - root directory of the Chainlink node
- CHAINLINKCONTAINERNAME - Chainlink node container name for the possibility of automating communication with it
- COMPOSEPROJECTNAME - Docker network project name for the possibility of automating communication with it. See the Docker Documentation to learn more.

If environment variables related to a Chainlink node, including a Link Token contract address, were changed during your work you should run the `make fct-run-nodes` command in order for them to be applied.

1. Give Forge permission to read the output directory of the toolkit by adding this setting to the `foundry.toml`:

```
fspermissions = [{ access = "read", path =  
"lib/foundry-chainlink-toolkit/out"}]
```

The default path to the root of the Foundry-Chainlink toolkit is `lib/foundry-chainlink-toolkit`. Unfortunately at the moment `foundry.toml` cannot read all environment variables. Specify a different path if necessary.

1. Incorporate the `makefile-external` into your project. To do this, create or update a `makefile` in the root of your project with:

```
-include ${FCTPLUGINPATH}/makefile-external
```

</Accordion>

<Accordion title="Set up chain RPC node" number={2}>

In order for a Chainlink node to be able to interact with the blockchain, and to interact with the blockchain using the Forge, you have to know an RPC node http endpoint and web socket for a chosen network compatible with Chainlink. In addition to the networks listed in this list, Chainlink is compatible with any EVM-compatible networks.

For local testing, we recommend using Anvil, which is a part of the Foundry toolchain.

You can run it using the following command:

```
make fct-anvil
```

If the local Ethereum node is restarted, re-initialize the Chainlink cluster or perform a clean spin-up of the Chainlink nodes to avoid possible synchronization errors.

</Accordion>

<Accordion title="Run the example" number={3}>

Scripts for automating the initialization of the test environment and setting up Chainlink jobs will be described below.

To display autogenerated help with a brief description of the most commonly used scripts, run:

```
make fct-help
```

For a more detailed description of the available scripts, you can refer to `DOCUMENTATION.md` in the repository.

</Accordion>

<Accordion title="Initialize testing environment" number={4}>

make fct-init

This command automatically initializes the test environment, in particular, it makes clean spin-up of a Chainlink cluster of 5 Chainlink nodes.

Once Chainlink cluster is launched, a Chainlink nodes' Operator GUI will be available at:

- http://127.0.0.1:6711 - Chainlink node 1
- http://127.0.0.1:6722 - Chainlink node 2
- http://127.0.0.1:6733 - Chainlink node 3
- http://127.0.0.1:6744 - Chainlink node 4
- http://127.0.0.1:6755 - Chainlink node 5

For authorization, you must use the credentials specified in the chainlinkapicredentials.

</Accordion>

<Accordion title="Set up Chainlink Jobs" number={5}>

make fct-setup-job

This command displays a list of available Chainlink jobs and sets up the selected one.

You can also set up a Chainlink job by calling the respective command. Manual set up of a Chainlink Job is recommended when using a custom Consumer or Aggregator contract, or when a different job configuration is desired. You can create a custom TOML file and use it to create a Chainlink Job instance through the Operator GUI or develop a custom script using the existing scripts provided by this toolkit. Manual setup steps are documented repository for each job type.

Direct Request Job

make fct-setup-direct-request-job

This command automatically sets up a Direct Request job.

Cron Job

make fct-setup-cron-job

This command automatically sets up a Cron job.

Webhook Job

make fct-setup-webhook-job

This command automatically sets up a Webhook job.

Keeper Job

```
make fct-setup-keeper-job
```

This command automatically sets up a Keeper job.

OCR Job

```
make fct-setup-ocr-job
```

This command automatically sets up an OCR job.

</Accordion>

Project Structure

You can find the full project structure in the repository.

```
# functions-demo-app.mdx:
```

```
---
title: "Chainlink Functions Demo App"
description: "Learn how to build applications that use Chainlink Functions by
running your own version of the Chainlink Functions Demo App."
image: "QuickStarts-Chainlink-Functions-Showcase.webp"
products: ["general"]
time: "90 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Aside } from "@components"
```

```
<Aside type="caution" title="Polygon testnet change">
The Mumbai network has stopped producing blocks, so example code will not
function on this network. Check again soon
for updates about future testnet support on Polygon.
</Aside>
```

Overview

The Chainlink Functions Demo App is designed to run on the Mumbai testnet (Polygon). It uses Chainlink Functions. The functionality allows users to donate POL to their favorite GitHub creators. Authors of those repositories can then claim their donations. Donations are made in an amount of POL per amount of Stars the repository has.

Chainlink Functions is used to determine the total donation amount by multiplying the POL amount by the star count. There's no back-end involved in the whole donation process.

```
<Aside type="caution" title="Disclaimer">
```

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how

to interact with Chainlink's systems and services so that you can integrate them into your own. This template is

provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key

checks or error handling to make the usage of the product more clear. Do not use the code in this example in a

production environment without completing your own audits and application of

best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Objective

You will create and manage a ledger contract to be used by the dApp. Tracking interaction between accounts gives a better insight into how the dApp functions. Using a different wallet for contract creation and dApp usage is preferable.

Before you begin

Before you start this tutorial, complete the following items:

- If you are new to smart contract development, learn how to Deploy Your First Smart Contract.
- Set up a cryptocurrency wallet such as MetaMask.
- To deploy this contract on testnets, ensure the deployer account has testnet ERC-677 LINK. Use the LINK faucet to retrieve testnet LINK.
 - Get testnet POL from faucet.polygon.technology to pay for your onchain transactions.
 - Get at least 2 ERC-677 testnet LINK from faucets.chain.link/mumbai.
- Install Node.js.
- Install pnpm: `npm install -g pnpm`
- Optional: If you want to verify your contracts onchain, create an account on Polygonscan and get an API key. Note that you'll need to create an account for the main network, which also works for the testnet.

Steps to implement

Run these from the project directory where you've cloned this repo.

1. Clone the repo and change directories:

```
bash
git clone https://github.com/smartcontractkit/chainlink-functions-demo-app.git && cd chainlink-functions-demo-app
```

1. Install the dependencies using pnpm:

```
bash
pnpm install
```

1. Create a .env file from the template:

```
bash
cp .env.example .env
```

1. In the .env file specify the following variables:

- PRIVATEKEY - Private key of your wallet used for deploying contracts.
- NEXTPUBLICGATRACKINGID - Optional - Set to your Google Analytics tracking id to enable GA.
- POLYGONSCANAPIKEY - Optional - API key for Polygonscan to verify and read contracts.

1. Generate and build all required files:

```
bash
```

```
pnpm build
```

1. Deploy the Ledger contract:

```
bash
npx hardhat project:deploy
```

1. The deployment script prints the deployed contract address to your terminal. Add the address in the NEXTPUBLICCONTRACTADDRESS environment variable to your .env file.

1. Optionally, you can verify the contract. This allows you to decode the bytecode on Polygonscan. Verify the contract with npx hardhat verify:

```
bash
npx hardhat verify --constructor-args arguments.js $NEXTPUBLICCONTRACTADDRESS
```

Replace \$NEXTPUBLICCONTRACTADDRESS with your contract address if you don't have the address in your shell environment.

1. Create a Chainlink Functions subscription and fund it using the project:fund script:

```
bash
npx hardhat project:fund
```

For more information on how to manage your subscription, read the official documentation.

1. Store the subscription ID in the NEXTPUBLICSUBSCRIPTIONID environment variable in your .env file.

Run the example

1. Run the dev server:

```
bash
pnpm dev
```

1. In the terminal output, you'll see a URL with a port number. Open this URL in your browser to see the UI running locally.

giveaway.mdx:

```
---
title: "Giveaway Manager"
description: "Build an app to manage distribution."
image: "QuickStarts-Giveaway-Manager.webp"
products: ["general"]
time: "180 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside, ClickToZoom, CopyText } from "@components"
import { TabsContent } from "@components/Tabs"
```

Overview

The Giveaway Manager app is a highly configurable proof of concept for provably-fair giveaways using Chainlink Automation and VRF. It is capable of drawing winners from a CSV list or onchain entries using VRF Direct Funding and Automation. Fulfillment is not included, and this app demonstrates provably fair selection only.

!image

Objective

In this tutorial, you will deploy a local user interface to enable giveaways using Chainlink VRFv2 Direct Funding. The UI was designed to run simple drawings and giveaways with just a CSV list of participants. Chainlink Automation provisioning and setup are also covered to enable timed-based, dynamic drawings.

You can use this to run:

- Static Giveaways: Fairly and transparently pick winners from a participant list in CSV format. Unique participant IDs can be anything (emails, numbers, addresses). They are hashed so no user data is stored onchain.
- Dynamic Giveaways: Create a giveaway that participants can enter on their own via tx, then fairly and transparently pick winners whenever you'd like. This allows participants to register over a set period of time without the admin needing a full participant list like a static giveaway.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided *AS IS* and *AS AVAILABLE* without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

<Aside type="note" title="New to smart contract development?">

If you are new to smart contract development, learn how to Deploy Your First Smart Contract.

</Aside>

Before you start this tutorial, complete the following items:

- Set up a cryptocurrency wallet such as MetaMask.
- Fund your wallet with the following testnet tokens:
 - Request ERC-677 testnet LINK and Sepolia ETH from faucets.chain.link/sepolia. You will need at least 5.1 LINK to set up each giveaway with this app.

Steps to implement

<Accordion title="Clone the repo" number={1}>

- Run `git --version` to check your git installation. You should see an output similar to `git version x.x.x`. If not, install git.
- Install NodeJS

- Install yarn

```
bash
git clone https://github.com/smartcontractkit/quickstart-giveaway.git
```

</Accordion>

<Accordion title="Create a block explorer API key" number={2}>

The recommended networks for this app are:

- Ethereum mainnet and Sepolia testnet

For demo purposes, use Sepolia for this tutorial. Create a block explorer API key to verify contracts on your preferred network:

- Etherscan

</Accordion>

<Accordion title="Set contract environment variables" number={3}>

1. Prepare the following values:

- The RPC URL for your deployment network, using an RPC node provider such as Infura or Alchemy.
- The private key for your deployer account. If your deployer account is in MetaMask, export your private key from MetaMask.
- The block explorer API key from Etherscan that you created earlier.

Parameter	Description
Example	

NETWORKRPCURL	The RPC URL for the network you want to deploy to.
https://eth-sepolia.g.alchemy.com/v2/your-api-key	
PRIVATEKEY	The private key of the account you want to deploy from.
 Add 0x before your key. 0xabc123abc123abc123abc123abc123...	
EXPLORERKEY	The block explorer API key needed for contract verification.
	ABC123ABC123ABC123ABC123ABC123ABC1

1. In the contracts directory, create a .env file for your contract environment variables:

```
bash
<root>/contracts
$ touch .env
$ open .env
```

If you're going to push this to your own repo, make sure this .env file is untracked, and consider using a secure env management package instead.

1. Add this content to your file and fill in the values. Do not use quotation marks when filling in the values. Add 0x to the beginning of your private key.

```
bash
Do NOT use quotes to assign values!
```

Network RPCs

```
export RPCURL=
```

```
Private key for contract deployment  
export PRIVATEKEY=
```

```
Explorer API key used to verify contracts  
export EXPLORERKEY=
```

</Accordion>

<Accordion title="Install Foundry" number={4}>

Refer to the Foundry installation instructions.

1. Download foundryup:

```
bash  
Download foundry  
$ curl -L https://foundry.paradigm.xyz | bash
```

1. Restart your terminal session, then install Foundry by running:

```
bash  
foundryup
```

<Aside type="note" title="MacOS installation note">

You might see the following error on MacOS: dyld: Library not loaded:
/usr/local/opt/libusb/lib/libusb-1.0.0.dylib. To fix this, install libusb:
brew install libusb See installation
troubleshooting for
details.
</Aside>

</Accordion>

<Accordion title="Install contract dependencies" number={5}>

Install GNU make if you do not already have it. The functionality of the project is wrapped in the makefile. Reference the below commands based on your OS or go to Make documentation.

macOS

1. The Xcode command line tools include make. If you've previously installed Xcode, skip to step 2 to verify your installation. Otherwise, open a Terminal window and run:

```
sh  
xcode-select --install
```

Alternatively, if you prefer to use Homebrew, be aware that GNU make is installed as gmake.

1. Verify your make installation:

```
sh  
make
```

If make is installed successfully, you will get the following output:

```
sh  
make: No targets specified or no makefile found. Stop.
```

Windows

1. If you're using WSL, open an Ubuntu terminal and run:

```
sh
sudo apt install make
```

1. Edit your path variable to include make.

1. Verify your make installation:

```
sh
make
```

If make is installed successfully, you will get the following output:

```
sh
make: No targets specified or no makefile found. Stop.
```

Install contract dependencies if changes have been made to contracts:

```
bash
<root>/contracts
$ make install
```

</Accordion>

<Accordion title="Deploy contract" number={6}>

```
bash
<root>/contracts
$ make deploy
```

Save the deployed contract address from your terminal output. This is the giveaway contract manager address that you will need when you run the UI. Scroll up in the terminal output and look for your contract address, which appears shortly after the Success message:

```
shell
sepolia
â€¦ [Success]Hash:
0x050ec798d7205c41bafa91029fcdd30104b99c17003a59ac033099dbe47d3658
Contract Address: 0xA168A5eAd28d5E4C1C9cEaF6492a0F9D715ea8D8
Block: 5727377
Paid: 0.005473916599834344 ETH (5470873 gas 1.000556328 gwei)
```

</Accordion>

<Accordion title="Install UI dependencies" number={7}>

1. To install the UI dependencies, navigate to the client directory and run:

```
bash
<root>/client
(Mac) you may need to run 'source ~/.nvm/nvm.sh'
nvm use
```

You may be prompted to install nvm and node v14.17.4. This process takes a few minutes.

1. After your node version is set, run:

```
bash
yarn
```

</Accordion>

<Accordion title="Run and view the UI" number={8}>

Set environment variables to run the UI and then view it locally.

1. Prepare your UI environment variables. These are needed for each time you want to run the UI:

Item	Value

UIGIVEAWAYMANAGERCONTRACTADDRESS	The address of the Giveaway Contract Manager contract that you deployed earlier
UILINKTOKENCONTRACTADDRESS	For Ethereum Sepolia: <CopyText text="0x779877A7B0D9E8603169DdbD7836e478b4624789" code/> See all LINK token contract addresses
UIKEEPERREGISTRYCONTRACTADDRESS	For Ethereum Sepolia: <CopyText text="0xE16Df59B887e3Caa439E0b29B42bA2e7976FD8b2" code/> This app currently supports Automation v1.2. See all Automation registry contract addresses

1. Navigate to the /client/packages/ui directory, and run these commands to set up your UI environment variables. Do not use quotes to assign values:

```
bash
<root>/client/packages/ui
export UIGIVEAWAYMANAGERCONTRACTADDRESS=
export UILINKTOKENCONTRACTADDRESS=0x779877A7B0D9E8603169DdbD7836e478b4624789
export
UIKEEPERREGISTRYCONTRACTADDRESS=0xE16Df59B887e3Caa439E0b29B42bA2e7976FD8b2
```

1. After setting the environment variables, run the UI locally:

```
bash
yarn start
```

1. To view the UI, open your browser at localhost:3005.

</Accordion>

<Accordion title="Implement a static giveaway" number={9}>

1. Navigate to http://localhost:3005/ in your browser.

1. In the upper right corner, click Connect wallet. Connect your wallet and ensure the proper network is selected.

<ClickToZoom src="/images/quickstarts/giveaway/1-ui-connect-wallet.png" alt="Connect wallet" />

1. Click Create giveaway.

<ClickToZoom src="/images/quickstarts/giveaway/2-ui-create-giveaway.png" alt="Create giveaway" />

1. Static giveaway is selected by default. Fill in the details to configure the giveaway:

- Create a CSV file with a list of giveaway participants. Below the

Participants field, download the example CSV participant file and fill it in. For testing, you can use a list of dummy wallet addresses.

- Fill in the rest of the fields and upload your CSV list of participants, then Create and confirm the wallet transaction that pops up.

```
{" "}
```

```
<ClickToZoom src="/images/quickstarts/giveaway/3-ui-create-static-giveaway.png" alt="Create static giveaway" />
```

1. After the transaction processes, you should get a "Giveaway successfully created" message. Click the UI card for the new giveaway you just created:

```
<ClickToZoom src="/images/quickstarts/giveaway/4-ui-click-giveaway.png" alt="Click the newly created giveaway" />
```

1. The details for your giveaway are displayed. Click Pick Winners and confirm the wallet transaction.

```
<ClickToZoom src="/images/quickstarts/giveaway/5-ui-pick-winners.png" alt="Pick winners" />
```

1. A "Pending Transaction" popup displays while the transaction is processing. Once the transaction is complete, you should see a "Successfully picked winners" message. Click Close.

- Optional: You can reference the "View VRF Request" link to help prove the giveaway was fair.

```
<ClickToZoom src="/images/quickstarts/giveaway/6-ui-pick-winners-confirmation.png" alt="Pick winners confirmation" />
```

1. At the bottom of the Giveaway page, upload your original participants (contestants) CSV again to see who won:

```
<ClickToZoom src="/images/quickstarts/giveaway/7-ui-upload-contestants-csv.png" alt="Upload contestants CSV" />
```

1. After you've uploaded the CSV, click Check Winners:

```
<ClickToZoom src="/images/quickstarts/giveaway/8-ui-check-winners.png" alt="Check winners" />
```

1. A Giveaway winners popup displays and shows the winners. This demo used a CSV of dummy wallet addresses. You can use whichever unique identifiers you want. Since the identifier is hashed before it's put onchain, the data remains private and visible only to you as the admin.

```
<ClickToZoom src="/images/quickstarts/giveaway/9-ui-giveaway-winners.png" alt="View winners" />
```

You can export a CSV list of all the winners and share the list with participants if you wish. You can also choose to provide the VRF transaction from Step 7 as evidence that Chainlink VRF was used for fairness.

</Accordion>

<Accordion title="Implement a dynamic giveaway" number={10}>

1. Navigate to <http://localhost:3005/> in your browser.

1. In the upper right corner, click Connect wallet. Connect your wallet and ensure the proper network is selected.

```
<ClickToZoom src="/images/quickstarts/giveaway/1-ui-connect-wallet.png"
```

alt="Connect wallet" />

1. Click Create giveaway.

<ClickToZoom src="/images/quickstarts/giveaway/2-ui-create-giveaway.png" alt="Create giveaway" />

1. For the Select giveaway type field, click Dynamic.

<ClickToZoom src="/images/quickstarts/giveaway/10-ui-select-dynamic-type.png" alt="Select giveaway type" />

1. Fill in the details to configure the giveaway:

- If you want the Giveaway to close automatically after a certain duration, click Enable Automation and enter the duration. If you don't do this, you must manually close the Giveaway before drawing.

- Click Create and confirm the wallet transaction that pops up.

<ClickToZoom src="/images/quickstarts/giveaway/11-ui-create-dynamic-giveaway.png" alt="Create dynamic giveaway" />

1. After the transaction processes, you should get a "Giveaway successfully created" message. Click the UI card for the new giveaway you just created:

<ClickToZoom
src="/images/quickstarts/giveaway/12-ui-click-dynamic-giveaway.png"
alt="Click the newly created giveaway"
/>

1. The details for your giveaway are displayed. At the bottom, there are a few options:

- Join Giveaway: Allows the admin to submit a tx to join the giveaway (useful for some use cases)

- Pick Winners: You can pick the winners at any time, if you specified a duration it will automatically close the giveaway so that nobody can enter anymore. You'll still need to run this to pick the winners.

- Cancel Giveaway: Cancel the giveaway and prevent anyone from entering.

<ClickToZoom src="/images/quickstarts/giveaway/13-ui-dynamic-giveaway-options.png" alt="Dynamic giveaway options" />

1. If you enabled a duration for the giveaway, you'll see the status change to Staged after the duration is complete. This means nobody else can join the giveaway and you're ready to pick winners.

<ClickToZoom src="/images/quickstarts/giveaway/14-ui-staged-status.png" alt="Staged status" />

1. The details for your giveaway are displayed. Click Pick Winners and confirm the wallet transaction.

<ClickToZoom src="/images/quickstarts/giveaway/15-ui-dynamic-pick-winners.png" alt="Pick winners" />

1. A "Pending Transaction" popup displays while the transaction is processing. Once the transaction is complete, you should see a "Successfully picked winners" message. Click Close.

- Optional: You can reference the "View VRF Request" link to help prove the giveaway was fair.

<ClickToZoom

```
src="/images/quickstarts/giveaway/16-ui-pick-winners-confirmation.png"
alt="Pick winners confirmation"
/>
```

1. After the transaction is confirmed, the Giveaway status changes to Finish. Click View Winners to see the wallet addresses of those who won.

```
<ClickToZoom src="/images/quickstarts/giveaway/17-ui-view-winners.png"
alt="View winners" />
```

You can export a CSV list of all the winners and share the list with participants if you wish. You can also choose to provide the VRF transaction as evidence that Chainlink VRF was used for fairness.

</Accordion>

Reference

Testing

To test contracts, navigate to the contracts directory and run the following command:

```
bash
<root>/contracts
make test-contracts-all
```

To test the UI, navigate to the /client/packages/ui directory and run the following commands:

```
bash
<root>/client/packages/ui
$ yarn test
$ yarn tsc
$ yarn lint
$ yarn prettier
```

Required balance amounts

As a creator of a giveaway, the minimum token requirements are needed to ensure that your giveaway is created and finished without issues. All unused LINK token amounts are able to be withdrawn after completion of giveaway.

- 5.1 LINK
 - 0.1 (VRF request)
 - 5 (Automation subscription)

Giveaway Status

After picking winners is initiated in the UI, the status of the giveaway is moved to pending. Each subsequent block is then checked to see if the VRF request has been finished and winners picked. Once found, the status is automatically moved to finished. The winners are then able to be viewed and leftover LINK is able to be withdrawn.

Developer Integration for Entering Dynamic Giveaway

The Giveaway contract is able to be integrated with any application that is able to send a transaction to the contract. The user will need to call the enterGiveaway function with the following parameters:

- giveawayId - The ID of the giveaway that the user is entering

- entries - The amount of entries the user is purchasing
- proof The merkle proof of the user's entry if the giveaway is permissioned

This is how the UI in this repo calls the enterGiveaway function using wagmi:

```
javascript
export const enterGiveaway = async (params: contracts.EnterGiveawayParams) => {
  try {
    const { id, proof, fee } = params
    const config = await prepareWriteContract({
      address: giveawayManagerContractAddress,
      abi: giveawayManagerABI,
      functionName: 'enterGiveaway',
      overrides: {
        value: ethers.utils.parseEther(fee)
      },
      args: [id, params.entries ? params.entries : 1, proof ? proof : []]
    })
    const data = await writeContract(config)
    return data
  } catch (error: any) {
    throw new Error(Error entering giveaway: ${error.message})
  }
}

export interface EnterGiveawayParams {
  id: number
  entries?: number
  proof?: string[]
  fee: string
}
```

hardhat-plugin.mdx:

```
---
title: "Using the Hardhat Chainlink Plugin"
description: "Learn how to use the Hardhat Chainlink Plugin in your applications."
image: "QuickStarts-Hardhat-Chainlink-Plugin.webp"
products: ["general"]
time: "30 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside } from "@components"
import { Tabs } from "@components/Tabs"
```

Overview

The Hardhat Chainlink Plugin offers a convenient way to integrate Chainlink functionality into your web3 development workflow by seamlessly interacting with Chainlink services within your Hardhat-based projects. Currently, the plugin supports Data Feeds, VRF, and Automation. It also provides a sandbox environment for running a local Chainlink node.

<Aside type="note" title="New to smart contract development?">

If you are new to smart contract development, learn how to Deploy Your First Smart

Contract. New to Hardhat? Check the Official Hardhat Documentation.

</Aside>

Objective

This tutorial shows you how to install and use the Hardhat Chainlink plugin.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

- Install NodeJS
- Install a supported package manager: either npm or yarn
- If you don't have an existing Hardhat project, create one by following the Hardhat Getting Started guide, and then return to this guide.

Installation

You can install the Hardhat Chainlink plugin using either npm or yarn. Choose the package manager that you prefer and run one of the following commands:

```
{/ prettier-ignore /}  
<Tabs client:visible>  
  <Fragment slot="tab.1">npm</Fragment>  
  <Fragment slot="tab.2">yarn</Fragment>  
  <Fragment slot="panel.1">  
    shell  
    npm install @chainlink/hardhat-chainlink  
  
  </Fragment>  
  <Fragment slot="panel.2">  
    shell  
    yarn add @chainlink/hardhat-chainlink  
  
  </Fragment>  
</Tabs>
```

After installation, add the plugin to your Hardhat config:

```
{/ prettier-ignore /}  
<Tabs client:visible>  
  <Fragment slot="tab.1">JavaScript</Fragment>  
  <Fragment slot="tab.2">TypeScript</Fragment>  
  <Fragment slot="panel.1">  
    In hardhat.config.js:  
    js  
    require("@chainlink/hardhat-chainlink");  
  
  </Fragment>  
  <Fragment slot="panel.2">  
    In hardhat.config.ts:  
    ts
```

```
import "@chainlink/hardhat-chainlink";
```

```
</Fragment>
```

```
</Tabs>
```

This plugin also extends the Hardhat configuration and adds chainlink parameters group in your config file:

```
ts
module.exports = {
  chainlink: {
    confirmations // Number of confirmations to wait for transactions, default:
1    1
  },
  ...
}
```

Choose how to use the plugin

The Hardhat Chainlink plugin offers multiple ways to interact with Chainlink services, giving you the flexibility to choose the approach that suits your workflow best:

Method of plugin usage	
Considerations	
<hr/>	

Command-line interface (CLI)	
Best option for beginners and simpler workflows	
Hardhat tasks: Call the plugin as a subtask within custom Hardhat tasks	
Complex workflows and automation; familiarity with using Hardhat tasks	
Methods in Hardhat environment: Call Chainlink services directly as methods	
Integrates well with an existing Hardhat project	

Choose the method that fits your project's requirements and coding style. All three approaches provide the same set of functionalities, allowing you to interact with Chainlink services efficiently and effectively.

CLI

Interact with the Hardhat Chainlink plugin through the command-line interface (CLI) using the following format:

```
npx hardhat chainlink:{service} [method] [--args]
```

This approach serves both as a CLI method and Hardhat tasks. However, it's important to note that the methods in each service are "hidden" with subtasks and won't be shown when you call `npx hardhat`. Instead, you can call the methods by passing its name as a parameter for the related task.

If the subtask and/or args are not passed directly, they will be interactively inquired during the CLI command execution.

<Aside type="note">

<p>Arguments for methods called with CLI should be provided as a valid JSON string.</p>

</Aside>

To get a list of all available methods (subtasks) and their arguments for a

specific service, you can use:

```
npx hardhat chainlink:{task}:subtasks
```

Example of calling a subtask with arguments directly in the CLI:

```
shell
npx hardhat chainlink:dataFeed getLatestRoundAnswer --args '{"dataFeedAddress":
"0xE62B71cf983019Bff55bC83B48601ce8419650CC"}'
```

Example of interacting with the CLI interactively, where the subtask and arguments are inquired:

```
shell
npx hardhat chainlink:dataFeed
The CLI will ask you to select a subtask (getLatestRoundAnswer) and provide
arguments interactively.
```

Hardhat tasks

Integrate the Hardhat Chainlink plugin as a subtask in your own Hardhat tasks. Use the following format to run a subtask:

```
hre.run("chainlink:{service}:{method}", { ...args });
```

This method is well-suited for more complex workflows and automation. You can call the {service} and {method} directly in your custom Hardhat tasks, passing the required arguments as an object containing the necessary parameters.

Example of calling a subtask in a custom Hardhat task with arguments:

```
js
task("myTask", "My custom task", async (taskArgs, hre) => {
  await hre.run("chainlink:dataFeed:getLatestRoundAnswer", {
    dataFeedAddress: "0xE62B71cf983019Bff55bC83B48601ce8419650CC",
  })
})
```

Methods in Hardhat environment

Directly access Chainlink services as methods in the Hardhat Environment using the following format:

```
hre.chainlink.{service}.{method}(...args);
```

This approach is ideal for seamless integration with your existing Hardhat project.

You can use familiar JavaScript syntax to call the Chainlink services as methods within your scripts and tasks.

Example of calling a subtask as a method in the Hardhat Environment:

```
js
async function myFunction() {
  const dataFeedAddress = "0xE62B71cf983019Bff55bC83B48601ce8419650CC"
```

```

    const result = await
hre.chainlink.dataFeed.getLatestRoundAnswer(dataFeedAddress)
    console.log(result)
}

```

Available services

The Hardhat Chainlink plugin supports the following Chainlink services:

- dataFeed (Data Feeds)
- dataFeedProxy (Data Feed Proxies)
- feedRegistry (Feed Registries)
- l2Sequencer (L2 Sequencers)
- ens (ENS - Ethereum Name Service)
- automationRegistry (Automation Registries)
- automationRegistrar (Automation Registrars)
- vrf (Verifiable Random Functions)

For a more in-depth understanding of available services and methods, please explore their tests.

Plugin registries

The Hardhat Chainlink plugin provides registries that contain information about smart contracts related to various Chainlink services and other data, such as the denominations library, which is useful for interacting with the Feed Registry.

In general, these registries help you access essential contract addresses deployed on different networks, making it easier to integrate Chainlink services into your projects.

Available registries

The Hardhat Chainlink plugin provides the following registries:

- dataFeeds: Addresses of Data Feeds-related contracts: Aggregators and Proxies, and their parameters.
- feedRegistries: Feed Registries' contract addresses.
- l2Sequencers: L2 Sequencer Uptime Feeds' contract addresses.
- keeperRegistries: Addresses of Automation-related contracts: Keeper Registry and Keeper Registrar.
- linkTokens: Link Tokens' contract addresses.
- vrfCoordinators: Addresses of VRF Coordinators and their parameters.
- denominations: Records from Denominations library to interact with Feed Registries contracts.

For a more in-depth understanding of the structure of these records, please explore their interfaces.

Using plugin registries

You can access the plugin registries using one of the following methods:

```

<Tabs client:visible>
<Fragment slot="tab.1">CLI</Fragment>
<Fragment slot="tab.2">Directly calling methods</Fragment>
<Fragment slot="panel.1">

```

To interact with the registries through the CLI, use the following command:

```

shell
npx hardhat chainlink:registries [method]

```

This command allows you to query records available in the registries. The CLI will inquire about the necessary additional information, such as the preferable network, to retrieve the required record from the registry.

To get a list of all available getter-method for a specific registry, you can use:

```
shell
npx hardhat chainlink:registries:subtasks
```

The CLI will also inquire about the registry getter-method interactively if not provided directly.

Example of getting a record from registry directly in the CLI:

```
shell
npx hardhat chainlink:registries getDataFeed
```

The CLI will ask you to select a preferred network and subsequent parameters.

</Fragment>

<Fragment slot="panel.2">

Access the registries as methods directly in the Hardhat Environment:

```
shell
const registry = hre.chainlink.registries.{registryName};
```

Replace {registryName} with the name of the registry. For example, dataFeeds, feedRegistries, or keeperRegistries.

Example of getting data from registry in the Hardhat Environment:

```
js
async function myFunction() {
  const dataFeedAddress =
hre.chainlink.registries.dataFeeds.ethereum.ETH.USD.contractAddress
  console.log(dataFeedAddress)
  // 0xE62B71cf983019BFf55bC83B48601ce8419650CC
}
```

</Fragment>

</Tabs>

historical-price-feeds-api.mdx:

```
---
title: "Historical Price Feeds API"
description: "Deploy your own Historical Price Feeds API for retrieving data."
image: "QuickStarts-Historical-Price-Feed.webp"
products: ["feeds"]
time: "10 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside } from "@components"
```

Overview

This API endpoint allows you to fetch historical prices from Chainlink price

feeds for a specified period. The API endpoint returns the price data in JSON format. You can use this API endpoint to fetch historical prices for a single round or multiple rounds.

Objective

This example shows you how to set up and run a pre-built API that reads historical price data from Chainlink Price Feeds. The API runs offchain, but reads the data from onchain Chainlink Data Feeds. You can use this API to build your own applications.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided `AS IS` and `AS AVAILABLE` without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

- Install Node.js `>= 14.x.x`
- Install Yarn `>= 1.22.x`

Steps to implement

<Accordion title="Set up the project" number={1}>

1. Clone the Historical Prices API repo and change directories:

```
bash
git clone https://github.com/smartcontractkit/quickstarts-historical-prices-api.git && cd quickstarts-historical-prices-api
```

1. Install the dependencies:

```
bash
yarn install
```

1. Build the project:

```
bash
yarn build
```

1. Start the server:

```
bash
yarn start
```

</Accordion>

<Accordion title="Write, compile, and deploy your first smart contract">

number={2}>

After you complete the setup steps, you can access the UI at <http://localhost:3000>. The API is available at <http://localhost:3000/api/price>.

In order to fetch the price for a single round, you need to specify the same start and end timestamps. Make sure that the time frame exists. The API endpoint will return the price for the round that is within the specified time frame.

Request a single-round

Make a simple curl request to test the API:

```
bash
curl http://localhost:3000/api/price?
contractAddress=0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419&startTimestamp=161455
6800&endTimestamp=1614556800&chain=mainnet&rpcUrl=https://eth-
mainnet.alchemyapi.io/v2/your-api-key
```

Response:

```
json
{
  "description": "ETH/USD",
  "decimals": 8,
  "rounds": [
    {
      "phaseId": "1",
      "roundId": "1",
      "answer": "2000",
      "timestamp": "2021-03-01T00:00:00Z"
    },
    {
      "phaseId": "1",
      "roundId": "2",
      "answer": "2100",
      "timestamp": "2021-03-01T01:00:00Z"
    }
  ]
}
```

</Accordion>

<Accordion title="Request multiple rounds" number={3}>

In order to fetch the prices for multiple rounds, you need to specify different start and end timestamps. Make sure that the start timestamp is less than the end timestamp and that the time frames exist. The API endpoint will return the prices for the rounds that are within the specified time frame.

Make a simple curl request to test the API with multiple rounds returned:

```
bash
GET /api/price?
contractAddress=0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419&startTimestamp=161455
6800&endTimestamp=1614643200&chain=mainnet&rpcUrl=https://eth-
mainnet.alchemyapi.io/v2/your-api-key
```

Example response:

```
json
```

```
{
  "description": "ETH/USD",
  "decimals": 8,
  "rounds": [
    {
      "phaseId": "1",
      "roundId": "1",
      "answer": "2000",
      "timestamp": "2021-03-01T00:00:00Z"
    },
    â€®
    â€®
  ]
}
```

</Accordion>

Reference

The endpoint for this API is GET /api/price.

Query Parameters

Required	Parameter	Description
Type	Options	
â€®	contractAddress	The address of the Chainlink price feed contract.
	string	Price Feeds
â€®	startTimestamp	The start timestamp of the period for fetching prices.
	number	Unix timestamp in seconds. Example: 1681187628
â€®	endTimestamp	The end timestamp of the period for fetching prices.
	number	Unix timestamp in seconds. Example: 1681187628
â€®	chain	The blockchain network where the contract is deployed.
	string	mainnet, arbitrum, bsc, polygon , avalanche, fantom, moonbeam, moonriver, optimism, metis, gnosis
	rpcUrl	The RPC URL for the blockchain network.
	string	RPC URLs

Response

The response is a JSON object with the following properties:

- description: The price pair name.
- decimals: The number of decimals for the answer.
- rounds: An array of round data objects. Each round data object has the following properties:
 - phaseId: The phase ID of the round.
 - roundId: The round ID.
 - answer: The price at the round.
 - timestamp: The timestamp of the round in UTC.

Errors

The API endpoint may return one of the following errors:

- Input validation error: This error is returned when the input parameters are not valid.
- Failed to get client for chain: This error is returned when the API fails to get a client for the specified blockchain network.
- Failed to get phase data from contract: This error is returned when the API fails to get phase data from the contract.

pass-cost-to-end-user.mdx:

```
---
title: "Pass VRF Costs to the End Users"
description: "Identify the cost of specific VRF requests in order to pass this cost to the end users of your application."
image: "QuickStarts-Pass-VRF-Costs-to-the-End-Users.webp"
products: ["vrf"]
time: "30 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside, ClickToZoom, CopyText } from "@components"
import { TabsContent } from "@components/Tabs"
```

Objective

The VRF direct funding method calculates costs at request time. In this tutorial, you will use VRF direct funding and map VRF request IDs to their corresponding responses from the VRF service, along with the cost of each request.

Before you begin

```
<Aside type="note" title="New to smart contract development?">
  If you are new to smart contract development, learn how to Deploy Your First
  Smart
  Contract.
</Aside>
```

Before you start this tutorial, complete the following items:

- Install Foundry.

```
{/ prettier-ignore /}
<TabsContent sharedStore="osType" client:visible>
  <Fragment slot="tab.1">macOS</Fragment>
  <Fragment slot="tab.2">Windows</Fragment>
  <Fragment slot="panel.1">
    1. Open a Terminal window and run:
      sh
      curl -L https://foundry.paradigm.xyz | bash

    1. Follow the instructions in your terminal to add foundryup to your CLI.
    1. Run foundryup. After successful installation, you should get output that
contains a Foundry banner. After that,
    you should see foundryup installing forge and other packages:
      sh
      foundryup: installing foundry (version nightly, tag nightly-
74c03182d41c75c40e5a1c398aca9400305ff678)
      foundryup: downloading latest forge, cast, anvil, and chisel
      100.0%
      foundryup: downloading manpages
      100.0%
      foundryup: installed - forge 0.2.0 (74c0318 2023-09-
14T00:27:34.321593000Z)
```

```
foundryup: installed - cast 0.2.0 (74c0318 2023-09-14T00:27:34.059889000Z)
foundryup: installed - anvil 0.2.0 (74c0318 2023-09-14T00:27:34.192518000Z)
foundryup: installed - chisel 0.2.0 (74c0318 2023-09-14T00:27:33.928662000Z)
foundryup: done!
```

</Fragment>

<Fragment slot="panel.2">

1. Foundry does not support Powershell or Cmd, so you will need to install and use WSL. (Git Bash for Windows also works with Foundry.)

1. After installing WSL or Git Bash, open a terminal window and run:

```
sh
curl -L https://foundry.paradigm.xyz | bash
```

1. Follow the instructions in your terminal to add foundryup to your CLI.

1. Run foundryup. After successful installation, you should get output that contains a Foundry banner. After that, you should see foundryup installing forge and other packages:

```
sh
foundryup: installing foundry (version nightly, tag nightly-74c03182d41c75c40e5a1c398aca9400305ff678)
foundryup: downloading latest forge, cast, anvil, and chisel
100.0%
foundryup: downloading manpages
100.0%
foundryup: installed - forge 0.2.0 (74c0318 2023-09-14T00:27:34.321593000Z)
foundryup: installed - cast 0.2.0 (74c0318 2023-09-14T00:27:34.059889000Z)
foundryup: installed - anvil 0.2.0 (74c0318 2023-09-14T00:27:34.192518000Z)
foundryup: installed - chisel 0.2.0 (74c0318 2023-09-14T00:27:33.928662000Z)
foundryup: done!
```

</Fragment>

</TabsContent>

- Install make if you do not already have it.

{/ prettier-ignore /}

<TabsContent sharedStore="osType" client:visible>

<Fragment slot="tab.1">macOS</Fragment>

<Fragment slot="tab.2">Windows</Fragment>

<Fragment slot="panel.1">

1. The Xcode command line tools include make. If you've previously installed Xcode, skip to step 2 to verify your installation. Otherwise, open a Terminal window and run:

```
sh
xcode-select --install
```

Alternatively, if you prefer to use Homebrew, be aware that GNU make is installed as gmake.

1. Verify your make installation:

```
sh
make
```

If make is installed successfully, you will get the following output:

```
sh
make: No targets specified or no makefile found. Stop.
```

</Fragment>

<Fragment slot="panel.2">

1. If you're using WSL, open an Ubuntu terminal and run:

```
sh
sudo apt install make
```

1. Edit your path variable to include make.
1. Verify your make installation:

```
sh
make
```

If make is installed successfully, you will get the following output:

```
sh
make: No targets specified or no makefile found. Stop.
```

</Fragment>
</TabsContent>

- Run `git --version` to check your git installation. You should see an output similar to `git version x.x.x`. If not, install git.
- Get your Etherscan API key for contract verification. If you do not already have one:
 - Create an Etherscan account.
 - Create an Etherscan API key.
- Create an account with Infura or Alchemy to use as an RPC endpoint if you do not already have one. Alternatively, you can use your own testnet clients.

Steps to implement

<Accordion title="Clone the example repo" number={1}>

Clone the `vrf-direct-funding-example` repo:

```
sh
git clone git@github.com:smartcontractkit/vrf-direct-funding-example.git
```

This gives you an existing Foundry project to configure and use.

</Accordion>

<Accordion title="Configure the Foundry project" number={2}>

Foundry is an Ethereum application development toolkit that is used here to configure and deploy the contract. You need the following information:

- The RPC URL for your deployment network, using an RPC node provider such as Infura or Alchemy.
- The private key for your deployer account. If your deployer account is in MetaMask, export your private key from MetaMask.
- An Etherscan API key for contract verification.

1. Create an `.env` file at the root of the `vrf-direct-funding-example/` directory:

```
Network RPCs
export RPCURL=
```

```
Private key for contract deployment
export PRIVATEKEY=
```

```
Explorer API key used to verify contracts
export ETHERSCANAPIKEY=
```

1. In your `.env` file, fill in these values:

Parameter	Description
Example	

RPCURL	The RPC URL for the network you want to deploy to.
https://sepolia.infura.io/v3/your-api-key	
PRIVATEKEY	The private key of the account you want to deploy from. Add 0x before your key.
0xabc123abc123abc123abc123abc123...	
EXPLORERKEY	The API key for Etherscan needed for contract verification.
ABC123ABC123ABC123ABC123ABC123ABC1	

</Accordion>

<Accordion title="Install dependencies" number={3}>

At the root of the vrf-direct-funding-example/ directory, run:

```
sh
make install
```

This command will install the project dependencies, which include chainlink and openzeppelin-contracts for secure smart contract development.

</Accordion>

<Accordion title="Deploy the contract" number={4}>

Deploy the contract

Foundry includes Forge, an Ethereum testing framework. The project's Makefile helps you invoke Forge with simple commands to start the deployment script.

At the root of the vrf-direct-funding-example/ directory, run this command to deploy the contract:

```
shell
make deploy
```

</Accordion>

<Accordion title="Run the example" number={5}>

Test the contract as the end-user.

1. Open the LINK Token Contract write functions for your network in an explorer. For example, on Ethereum Sepolia you can open the 0x779877A7B0D9E8603169DdbD7836e478b4624789 contract.

1. Connect your wallet to the block explorer (e.g. Etherscan for Ethereum Sepolia) so you can run write functions.

```
<ClickToZoom
  src="/images/quickstarts/pass-cost-to-end-user/connect-to-scanner.webp"
  alt="Connect wallet to the block explorer"
  style="max-width: 50%;"
/>
```

1. Approve the deployed contract to spend LINK. Run the approve function with your deployed contract address and <CopyText text="5000000000000000000" code/> Juels (5 LINK) as variables. Click Write to run the function. MetaMask asks you to approve the transaction.

```

<ClickToZoom
  src="/images/quickstarts/pass-cost-to-end-user/approve-link-spend.webp"
  alt="Approve LINK spend"
  style="max-width: 50%;"
/>

```

1. Open your deployed contract in the block explorer.

1. On the Contract tab, open the Write Contract functions list for your deployed contract.

1. Again, connect your wallet to the block explorer so you can run write functions on the deployed contract.

```

<ClickToZoom
  src="/images/quickstarts/pass-cost-to-end-user/connect-to-scanner-deployed-
contract.webp"
  alt="Connect to the block explorer while viewing your deployed contract"
  style="max-width: 50%;"
/>

```

1. In the requestRandomWords function, click Write to initiate a request for randomness as an end-user. Confirm the transaction in MetaMask.

```

<ClickToZoom
  src="/images/quickstarts/pass-cost-to-end-user/request-randomness.webp"
  alt="Request randomness with the Write button under requestRandomWords"
  style="max-width: 50%;"
/>

```

1. If the transaction is successful, you will see a View your transaction button.

1. On the Read Contract tab for your deployed contract, view and copy lastRequestId value.

1. Use this value to find the request status and view the random value. Expand getRequestStatus, enter the requestId, and click Query. The randomWords value displays in the response below the Query button.

```

<ClickToZoom
  src="/images/quickstarts/pass-cost-to-end-user/request-status-with-
answer.webp"
  alt="View the request status with your lastRequestId value"
  style="max-width: 50%;"
/>

```

</Accordion>

Code example

This section explains how the VRFDirectFundingConsumer.sol contract maps each request to its cost.

This RequestStatus struct is used to map attributes of each VRF request to its request ID, including its cost, its fulfillment status, and the random values generated by the VRF service in response to the request:

```

solidity
struct RequestStatus {
    uint256 paid;
    bool fulfilled;
    uint256[] randomWords;
}

```

```
}
```

```
mapping(uint256 => RequestStatus) public srequests;
```

The requestRandomWords function calls the calculateRequestPrice function on the VRF wrapper contract to calculate the request cost upfront, and then transfers enough LINK to the VRF wrapper to cover the request. After sending the request, the requestRandomWords function stores the RequestStatus with the cost and other request attributes, and emits a RequestSent event.

```
solidity
function requestRandomWords() external returns (uint256) {
    // Calculate the price required for the VRF request based on the configured
    parameters.
    uint256 requestPrice = calculateRequestPrice();

    // Transfer the required LINK tokens from the caller's address to this
    contract.
    transferLinkFromUser(requestPrice);

    // Send the randomness request to the VRF Wrapper contract and retrieve the
    request ID and the paid price.
    bytes memory extraArgs =
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment:
false}));
    (uint256 requestId, uint256 reqPrice) = requestRandomness(
        requestConfig.callbackGasLimit, requestConfig.requestConfirmations,
requestConfig.numWords, extraArgs
    );
    // Record the request details.
    srequests[requestId] = RequestStatus({paid: reqPrice, randomWords: new
uint256[](0), fulfilled: false});
    requestIds.push(requestId);
    lastRequestId = requestId;

    // Emit an event indicating that a request has been sent.
    emit RequestSent(requestId, requestConfig.numWords);

    return requestId;
}

[...]

function calculateRequestPrice() internal view returns (uint256) {
    return
ivrfV2PlusWrapper.calculateRequestPrice(requestConfig.callbackGasLimit,
requestConfig.numWords);
}
```

After the VRF service fulfills your randomness request, it calls back your fulfillRandomWords function, where you implement logic to handle the random values. It's recommended to keep the processing in your callback function minimal to save on gas costs. In this case, the example contract updates the RequestStatus in the request mapping, and emits an event indicating that the request has been fulfilled. Another part of your application can listen for this event and further process the random values separately from the callback function.

```
solidity
function fulfillRandomWords(uint256 requestId, uint256[] memory randomWords)
internal override {
    require(srequests[requestId].paid > 0, "request not found");
```

```

    srequests[requestId].fulfilled = true;
    srequests[requestId].randomWords = randomWords;
    emit RequestFulfilled(requestId, randomWords, srequests[requestId].paid);
}

```

price-feeds-showcase.mdx:

```

---
title: "Solana Pricefeeds Showcase"
description: "Learn how to deploy a Price Feeds app for Solana."
image: "QuickStarts-Solana-Pricefeeds-Showcase.webp"
products: ["feeds"]
time: "10 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---

```

```
import { Accordion, Aside } from "@components"
```

Overview

You can use onchain Chainlink Data Feeds to demonstrate how to build a simple prediction game using Solana.

Objective

This tutorial shows you how to install and configure the Solana Prediction Game locally so you can learn from this example and build your own applications on Solana that use Chainlink Data Feeds. You will set up your Solana environment, configure Vercel, and configure a MongoDB database.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

1. Install [Node.js 14 or higher][node.js-url]. Run `node --version` to verify which version you have installed:

```

sh
node --version

```

1. Set up yarn package manager.

```

sh
npm i -g yarn

```

1. Set up gh on your machine

```
sh
npm install -g gh
```

1. Fork and clone the repo

```
sh
gh repo fork https://github.com/smartcontractkit/solana-prediction-game.git
--clone
```

1. Set up Solana CLI

- MacOS & Linux
- Windows

1. Run solana --version to make sure the Solana CLI is installed correctly.

```
sh
solana --version
```

1. Set up Vercel

1. To install the latest version of Vercel CLI, run this command:

```
sh
npm i -g vercel
```

1. To quickly start a new project, run the following commands:

```
sh
cd solana-prediction-game    Change directory to the project
vercel                      Deploy to the cloud
```

1. Connect your Git repository to Vercel and deploy with git push.

1. Set up a Solana wallet. For this example, install the Phantom wallet:

1. Visit <https://phantom.app/download> and select your browser type. Follow the steps in your respective extension store to add Phantom to your browser. After installing, you should see Phantom start-up in a new tab.

1. If you are a brand new Solana user, select "Create New Wallet" and create a password. If you are an existing Solana user, you can follow the steps here to migrate your existing wallets to Phantom.

1. Store your "Secret Recovery Phrase" in a safe and secure location, it is the only way to recover your wallet. Whoever has access to this phrase has access to your funds.

1. Set up Phantom mobile app.

1. Visit <https://phantom.app/download> and select the app marketplace according to your device type (iOS/Android). You will be redirected to the Phantom application on your device app store, follow the steps to download and install the application on your device. After installing, open the app and you should see the Phantom onboarding screen to set up a new wallet.

1. If you are a brand new user, select "Create a new wallet" and enable the device authentication (which may vary according to the device type) to protect your wallet from unauthorized access. If you are an existing Solana user, you can select "I already have a wallet" and enter your seed phrase to restore your wallets.

1. You can add additional security, e.g: enabling bio-metric security or face recognition, in Settings (Upper left corner), Security & Privacy.

1. Switch to Solana Devnet

1. Open your Phantom Wallet, click on the gear icon in the bottom right-hand corner, scroll to Change Network and click on Devnet.

1. Airdrop yourself Devnet Solana Tokens.

1. Open the faucet: <https://solfaucet.com/>

1. Copy your Phantom Wallet address

1. Paste your wallet address into the text field and then click 'DEVNET' to send the Solana token

1. Switch your network to devnet on your wallet.

Steps to implement

<Accordion title="Configure the application" number={1}>

1. Change to the project directory `cd solana-prediction-game`

1. Run `cp .env.example .env`

1. Install the NPM packages

`sh`

`yarn`

1. Download a wallet extension on your browser, preferably Phantom. Follow the instructions to set up your wallet.

1. You need an Escrow account to hold and pay out Solana to your users. There are two ways to acquire this:

1. Copy your private key from your Wallet. In this example, we will be using Phantom.

1. Open the Phantom browser extension menu

1. Create a new account

1. Select the menu and pick Security and Privacy

1. Then Export Private Key

1. Enter your password.

1. Copy the contents to `WALLETPRIVATEKEY=` in `.env`

1. Create a temporary Solana wallet via `solana-keygen`. Alternatively, if you have an existing wallet that you want to use, locate the path to your `[keypair]` `[keypair-url]` file and use it as the keypair for the rest of this guide.

`sh`

`solana-keygen new --outfile ./id.json`

Copy the contents of the array in `./id.json` to `WALLETPRIVATEKEY=` in `.env`

</Accordion>

<Accordion title="Set up a MongoDB database" number={2}>

1. Set up a MongoDB account via the following tutorial: [Create MongoDB Account](#).

1. Set up MongoDB cluster. [Create Cluster](#)

1. Set up MongoDB User. [Create User](#)

!Users Page

!Create User

1. Get the MongoDB Connection URI.

1. Navigate to the Database Deployments page for your project.

1. Click the Connect button.

!Connect

1. Select Connect your application

!Connection String

1. Copy the connection URI string to MONGODBURI= in your .env.
!URI string
1. Replace the <user> and <password> to your MONGODBURI with the username & password you created.
!MONGODBURI

1. Configure your Network Connection for your local development and deployment

- For Local development.

1. On the left sidebar of your screen select Network Access
1. Click on Add IP Address.
1. Click Add My Current IP Address.
1. Click Confirm and your IP address is added to the list of addresses.

- For Vercel deployment.

1. On the left sidebar of your screen select Network Access.
1. Click on Add IP Address.
1. In the box labeled Access List Entry add 0.0.0.0/0.
1. In the box labeled Required for your Cluster(s) linked to Vercel add Required for your Cluster(s) linked to Vercel.
1. Click Confirm and your IP address is added to the list of addresses.

!Network Access

</Accordion>

<Accordion title="Run the example" number={3}>

1. Generate a random API key here and copy it to APISECRETKEY=.

1. Run the following command to start the application:

```
sh
yarn development
```

You can access the app at the URL provided by Vercel.

!UI Screenshot

</Accordion>

time-based-upkeep.mdx:

```
---
title: "Time-based upkeeps"
description: "Time-based upkeeps are contracts you set up on Chainlink
Automation that allow you to run a smart contract function automatically
according to a custom schedule that you define. Time-based upkeeps are similar
to cron jobs."
image: "QuickStarts-Time-based-upkeep.webp"
products: ["automation"]
time: "15 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside, ClickToZoom, CodeSample } from "@components"
import { TabsContent } from "@components/Tabs"
```

Objective

In this tutorial, you will learn how to set up a time-based upkeep to automate a smart contract function using Chainlink Automation. You can use time-based upkeeps to:

- Schedule a rebalancing activity
- Schedule a recurring payment
- Schedule circuit breaker checks
- Automate triggering of any smart contract function that needs to run on a certain time schedule, or at a certain time in the future

Before you begin

Before you start this tutorial, complete the following items:

- If you are new to smart contract development, learn how to [Deploy Your First Smart Contract](#).
- Set up a cryptocurrency wallet such as MetaMask.
- Fund your wallet with the following testnet tokens:
 - The gas currency for the chain where your contract is deployed. For this example, get 0.5 Sepolia ETH from one of the available public faucets.
 - ERC-677 LINK. Get 20 testnet LINK.

Steps to implement

<Accordion title="Deploy a smart contract to automate" number={1}>

If you already have a deployed smart contract that you want to automate, make sure the contract has an external or public function that modifies the state of the contract onchain. In Solidity, pure and view functions do not update the state, so even if these functions are public, you cannot automate them with Chainlink Automation.

If you do not already have a deployed contract, follow these instructions to deploy a simple counter contract. This contract has a public counter variable that you can update by adding the updateAmount value to it.

1. Open the Counter.sol contract in Remix:

```
{/ prettier-ignore /}  
<CodeSample src="samples/Automation/Counter.sol" />
```

1. Under the Solidity compiler tab, compile the contract:

<ClickToZoom src="/images/automation/qs-time-based-upkeep/compile-counter.png" alt="Compile contract" />

1. Under the Deploy and run transactions tab, select Injected Provider - MetaMask for the Environment field.

<ClickToZoom src="/images/automation/qs-time-based-upkeep/injected-provider.png" alt="Remix Environment field" />

1. Click Deploy. MetaMask opens and prompts you to confirm the contract deployment transaction:

```
<ClickToZoom  
  src="/images/automation/qs-time-based-upkeep/metamask-contract-  
  deployment.png"  
  alt="Confirm MetaMask contract deployment transaction"  
/>
```

1. Copy the address of your newly deployed contract. You can find this in Sepolia Etherscan through the contract deployment transaction, or in Remix in the Deployed Contracts section:

```
<ClickToZoom
  src="/images/automation/qs-time-based-upkeep/metamask-contract-
deployment.png"
  alt="Confirm MetaMask contract deployment transaction"
/>
```

</Accordion>

<Accordion title="Provide contract information" number={2}>

If your smart contract is already verified onchain, skip to the next step.

Chainlink Automation needs your smart contract's Application Binary Interface (ABI) in order to automate the public functions in your contract. If your smart contract is not verified onchain, you must either provide your contract's ABI manually, or verify the contract onchain.

Verify your contract

Verify your contract directly in the blockchain explorer by adding your contract's source code and other required information in the blockchain explorer UI. For example:

- Verify contracts deployed to Sepolia on the testnet version of Etherscan.

After you successfully verify your smart contract, the Chainlink Automation App fetches its ABI automatically when you register an upkeep:

```
<ClickToZoom src="/images/automation/qs-time-based-upkeep/abi-fetched.png"
alt="ABI fetched automatically" />
```

Add the ABI manually

If you do not want to verify the contract on chain, paste the contract's ABI into the corresponding text box in the Chainlink Automation App while you are registering the upkeep.

1. In Remix, within the Solidity Compiler tab, copy the ABI here:

```
<ClickToZoom src="/images/automation/qs-time-based-upkeep/get-abi-remix.png"
alt="Remix Copy ABI" />
```

1. In the Chainlink Automation App, paste the ABI into the ABI field:

```
<ClickToZoom
  src="/images/automation/qs-time-based-upkeep/couldnt-fetch-abi-ui.png"
  alt="Chainlink Automation App ABI field"
/>
```

</Accordion>

<Accordion title="Register a new Upkeep" number={3}>

```
<div class="remix-callout">
  <a href="https://automation.chain.link">Open the Chainlink Automation App</a>
</div>
```

1. Click the Register New Upkeep button

```
<ClickToZoom src="/images/automation/auto-ui-home.png" alt="Click Register
New Upkeep" />
```

1. Connect your wallet using the Connect Wallet choose a network. For a list of

supported networks, see the Supported Blockchain Networks section. The Chainlink Automation App also lists the currently supported networks.

```
<ClickToZoom src="/images/automation/auto-ui-wallet.png" alt="Connect With Metamask" />
```

1. Select the time-based trigger.

1. Provide the address of your deployed contract. If your contract is not verified, paste the ABI into the corresponding text box.

1. If you're using the Counter contract, add a value for the function input in the Chainlink Automation App:

```
<ClickToZoom src="/images/automation/qs-time-based-upkeep/update-amount.png" alt="Function input" />
```

Each time Automation triggers your function, based on the schedule you give it, it will run the updateCounter function which increments the counter variable using the value you enter for updateAmount.

```
</Accordion>
```

```
<Accordion title="Specify the time schedule" number={4}>
```

After you enter the basic contract information, specify your time schedule in the form of a CRON expression. CRON expressions are a shorthand way to create a time schedule. Use the provided example buttons to experiment with different schedules and then create your own, or see the format and examples below.

For the Counter contract, enter `/5` to run the updateCounter function every five minutes:

```
<ClickToZoom src="/images/automation/qs-time-based-upkeep/time-schedule.png" alt="Time schedule example" />
```

The Chainlink Automation App displays the next five scheduled events to help you confirm that you have entered your desired schedule correctly.

After you enter your cron expression, click Next.

Cron format

Cron jobs are interpreted according to this format:

```
â"€â" â" â" â" â" â" â" â" â" â" â" â" minute (0 - 59)
â", â"â" â" â" â" â" â" â" â" â" â" â" hour (0 - 23)
â", â", â"â" â" â" â" â" â" â" â" â" day of the month (1 - 31)
â", â", â", â"â" â" â" â" â" â" â" â" month (1 - 12)
â", â", â", â", â"â" â" â" â" â" â" â" â" day of the week (0 - 6)
(Sunday to Saturday)
â", â", â", â", â",
â", â", â", â", â",
â", â", â", â", â",
```

All times are in UTC

- can be used for range e.g. "0 8-16 "

/ can be used for interval e.g. "0 /2 "

, can be used for list e.g. "0 17 0,2,4"

Special limitations:

- there is no year field
- no special characters: ? L W #
- lists can have a max length of 26
- no words like JAN / FEB or MON / TUES

Cron schedule examples

Cron expression	Schedule
*	Every minute
0 8-16	Every hour between 08:00-16:00 UTC
0 /2	Every two hours
0 17 0,2,4	Sunday, Tuesday and Thursday at 17:00 UTC

</Accordion>

```
<Accordion title="Entering upkeep details" number={5}>
```

To complete the upkeep registration process, you must enter some information about your upkeep including its name, gas limit, starting balance LINK, and contact information.

!Automation Upkeep Details

1. Complete the required details:

- Upkeep name: This will be publicly visible in the Chainlink Automation app.
- Gas limit: This is the maximum amount of gas that your transaction requires to execute on chain. This value defaults to 500000, which is sufficient for the simple updateCounter function in the Counter contract. If the gas required to execute your transaction exceeds the gas limit that you specified, your transaction will not be confirmed.
- Starting balance (LINK): Specify a LINK starting balance to fund your upkeep. If you're using the Counter contract, 3 LINK is a sufficient starting balance. If your upkeep is underfunded after you create it, you can add more funds later.
- Your email address: This email address will be encrypted and is used to send you an email when your upkeep is underfunded.

</Accordion>

```
<Accordion title="Check your counter variable" number={6}>
```

You can check the value of the counter variable in your contract in Remix. If you verified your contract, you can also check it in Sepolia Etherscan.

```
<TabsContent sharedStore="remixOrEtherscan" client:visible>
```

```
<Fragment slot="tab.1">Remix</Fragment>
```

<Fragment slot="tab.2">Sepolia Etherscan (Verified contract only)</Fragment>

```
<Fragment slot="panel.1">
```

In the Deployed Contracts tab, click the counter button to check the value of the counter variable.

```
<ClickToZoom src="/images/automation/qs-time-based-upkeep/remix-check-counter.png" alt="Check counter variable" />
```

</Fragment>

```
<Fragment slot="panel.2">
```

If you verified the Counter contract, you can check the value of the counter variable in Sepolia Etherscan to see that it has been updated after 5 minutes. For example, this contract has the following value after being updated every 5 minutes for about one day:

```
<ClickToZoom src="/images/automation/qs-time-based-upkeep/check-counter.png"
alt="Check counter variable" />
</Fragment>
</TabsContent>
```

```
</Accordion>
```

```
<Accordion title="Manage your upkeep" number={7}>
```

You can manage your upkeep in the Chainlink Automation App, or by directly using the registry contract functions.

In the Chainlink Automation App, within the Details page for each upkeep, you can find this Actions menu:

```
<ClickToZoom src="/images/automation/qs-time-based-upkeep/upkeep-actions-
menu.png" alt="Upkeep actions menu" />
```

To pause your upkeep, select Pause upkeep. Chainlink Automation App prompts you to approve the upkeep pause, while MetaMask opens and prompts you to confirm the "pause upkeep" transaction:

```
<ClickToZoom src="/images/automation/qs-time-based-upkeep/approve-upkeep-
pause.png" alt="Approve upkeep pause" />
<ClickToZoom src="/images/automation/qs-time-based-upkeep/metamask-pause-
upkeep.png" alt="MetaMask pause upkeep" />
```

Pausing the upkeep can be useful for debugging. If you discover a problem with the contract you registered initially, you can pause the upkeep, update the upkeep to use a new contract, and then unpause the upkeep.

For the upkeep to continue running, you must maintain a minimum balance. The upkeep stops running if it is underfunded. To withdraw funds from an upkeep, you must cancel the upkeep.

```
</Accordion>
```

```
# vrf-enabled-lootbox-pack.mdx:
```

```
---
title: "VRF-Enabled LootBox/Pack Contract"
description: "Build a pack contract using VRF."
image: "QuickStarts-VRF-Enabled-LootBox-Pack-Contract.webp"
products: ["vrf"]
time: "90 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Aside, CodeSample } from "@components"
import { Tabs } from "@components/Tabs"
```

Overview

The lootbox utility contract uses Chainlink VRF to determine a random reward that you can distribute as a Loot Box to users. This allows you to issue rewards fairly and transparently using verified randomness that the recipients can verify onchain.

```
{/ prettier-ignore /}
<div class="remix-callout">
  <a href="https://github.com/smartcontractkit/quickstarts-lootbox">See the code
on GitHub</a>
</div>
```

Objective

This tutorial demonstrates how to create a reward distribution dApp to distribute in-game rewards to players. In this tutorial, you will use example code from the `smartcontractkit/quickstarts-lootbox` repository. You will configure the example to work on a testnet of your choice, use an RPC endpoint of your choice, and distribute in-game rewards to players by using a lootbox contract. This example uses the HardHat framework to help you complete the testing and deployment steps.

This example contract supports distributing any rewards that adhere to fungible, non-fungible, and other configuration standards. For example, you can create a set of in-game rewards, transfer them to the lootbox contract when you deploy it, and then the lootbox contract manages in-game reward distributions.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how

to interact with Chainlink's systems and services so that you can integrate them into your own. This template is

provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key

checks or error handling to make the usage of the product more clear. Do not use the code in this example in a

production environment without completing your own audits and application of best practices. Neither Chainlink Labs,

the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due

to errors in code.

</Aside>

Before you begin

<Aside type="note" title="New to smart contract development?">

If you are new to smart contract development, learn how to [Deploy Your First Smart Contract](#).

</Aside>

Before you start this tutorial, complete the following items:

- Install git
 - Run `git --version` to check the installation. You should see an output similar to `git version x.x.x`.
- Install Nodejs 16.0.0 or higher
 - Run `node --version` to check the installation. You should see an output similar to `v16.x.x`.
 - Optionally, you can also Install Yarn and use it to run this example instead of using npm.
- Create an Etherscan API key if you do not already have one.
- Create an account with Infura or Alchemy to use as an RPC endpoint if you do not already have one. Alternatively, you can use your own testnet clients.

Your deployer account is the main account that you will use to deploy the lootbox contract, distribute rewards, pay for transaction fees, and pay for VRF costs. This is also the owner account.

- Ensure the deployer account owns all the assets you wish to distribute as rewards.
- To deploy this contract on testnets, ensure the deployer account has testnet LINK and testnet ETH (Sepolia). Use the LINK faucet to retrieve 20 testnet LINK and 0.1 Sepolia ETH.
- Ensure the deployer account has enough LINK to fund the VRF subscription. If

you don't already have a VRF subscription, the deployment script will create and fund one for you.

- Estimate the minimum subscription balance that VRF requires to process your randomness request. The minimum subscription balance provides a buffer against gas volatility, and only the actual cost of your request will be deducted from your account. If your subscription is underfunded, your VRF request will be pending for 24 hours. If that happens, check the Subscription Manager to see the additional balance needed.

- The initial funding amount is set to 10 LINK. Optionally, you can modify the initial funding amount in `network-config.ts`.

- Create a second account in your MetaMask wallet. This account is called the receiving account in this tutorial, and its address is called the opener address. You will use this receiving account to open a test lootbox and receive the rewards as a user. Configure the account to display the rewards that you want to distribute from the lootbox.

Steps to implement

<Accordion title="Clone the example repo and install dependencies" number={1}>

Clone the repo and install all dependencies.

If you want to use npm, run:

```
bash
git clone git@github.com:smartcontractkit/quickstarts-lootbox.git && \
cd quickstarts-lootbox && \
npm install
```

Alternatively, you can use yarn to install dependencies:

```
bash
git clone git@github.com:smartcontractkit/quickstarts-lootbox.git && \
cd quickstarts-lootbox && \
yarn install
```

</Accordion>

<Accordion title="Configure your project" number={2}>

Copy the `.env.example` file to `.env` and fill in the values.

```
bash
cp .env.example .env
```

</Accordion>

<Accordion title="Configure the Hardhat project" number={3}>

Hardhat is an Ethereum development environment that is used here to configure and deploy the lootbox contract. You need the following information:

- The RPC URL for your deployment network, using an RPC node provider such as Infura or Alchemy.
- The private key for your deployer account. If your deployer account is in MetaMask, export your private key from MetaMask.
- An Etherscan API key for contract verification. Create an Etherscan API key if you do not already have one.

1. In your `.env` file, fill in these values:

Parameter	Description
Example	

NETWORKRPCURL	The RPC URL for the network you want to deploy to.
https://sepolia.infura.io/v3/your-api-key	
PRIVATEKEY	The private key of the account you want to deploy from.
0xabc123abc123abc123abc123abc123...	
ETHERSCANAPIKEY	The API key for Etherscan needed for contract verification.
ABC123ABC123ABC123ABC123ABC123ABC1	

1. If you're using a deployment network other than Sepolia or Ethereum: configure your deployment network in both the .env file and in the hardhat.config.ts file.

</Accordion>

<Accordion title="Set your contract parameters" number={4}>

In your .env file, fill in these values:

Parameter	Description
Example	

LOOTBOXFEEPEROPEN	The fee per open in ETH
0.05	
LOOTBOXAMOUNTDISTRIBUTEDPEROPEN	The amount of reward units distributed per open
1	
LOOTBOXOPENSTARTTIMESTAMP	The start timestamp in UNIX time for the public open. Leave blank to start immediately.
16300000000	
VRFSUBSCRIPTIONID	A funded Chainlink VRF subscription ID. If you leave this blank, a new subscription will be created and funded on deploy.
7123	

The LOOTBOXFEEPEROPEN allows you to pass transaction and VRF costs to the user when they open the lootbox.

</Accordion>

<Accordion title="Define the rewards" number={5}>

You can distribute multiple types of in-game rewards using this lootbox contract. The lootbox contract supports any rewards of the following token standards:

- ERC-20: the fungible token standard
- ERC-721: the NFT standard
- ERC-1155: the multi-token standard

The amounts per unit are used to calculate the lootbox supply of rewards. For example, if you want to distribute 100 of a specific NFT, you would set the totalAmount to 10000000000000000000 and the amountPerUnit to 10000000000000000000. This would result in a lootbox supply of 100 units. For testing, you can distribute smaller amounts.

Add each type of in-game reward you want to distribute by following the format below.

<Aside type="note">

The deployer account must own the entire lootbox supply that you want to distribute as rewards.

</Aside>

There's an example configuration file in scripts/data/tokens.json which includes a list of all supported token types:

```
{/ prettier-ignore /}
<Tabs client:visible>
<Fragment slot="tab.1">ERC-20</Fragment>
<Fragment slot="tab.2">ERC-721</Fragment>
<Fragment slot="tab.3">ERC-1155</Fragment>
<Fragment slot="panel.1">
json
{
  "tokenType": "ERC20",
  "assetContract": "0x0000000000000000000000000000000000000000000000000000000000000001",
  "totalAmount": "10000000000000000000",
  "amountPerUnit": "10000000"
}
</Fragment>
<Fragment slot="panel.2">
json
{
  "tokenType": "ERC721",
  "assetContract": "0x0000000000000000000000000000000000000000000000000000000000000002",
  "tokenIds": ["1", "2"]
}
</Fragment>
<Fragment slot="panel.3">
json
{
  "tokenType": "ERC1155",
  "assetContract": "0x0000000000000000000000000000000000000000000000000000000000000003",
  "tokenId": "0",
  "totalAmount": "100",
  "amountPerUnit": "10"
}
</Fragment>
</Tabs>

</Accordion>
```

<Accordion title="Create an allowlist for private mints" number={6}>

The Merkle tree for the private opening mode is generated from the address list in the scripts/data/whitelist.json file. Edit the file and add all the addresses you want to list.

For this tutorial, add the address for your secondary account that you'll use to receive the rewards when you test opening the lootbox.

If you don't want to do a private mint, leave the whitelist.json file empty, and the contract will be initialized in public opening mode.

After deployment, you can optionally change some contract parameters by calling the following functions from the owner account:

Function	Description	Parameters
----------	-------------	------------

Function	Description	Parameter
setWhitelistRoot	Set new Merkle root for the allow list.	whitelistRoot
setPrivateOpen	Enable/disable public opening mode.	
privateOpenEnabled		

</Accordion>

<Accordion title="Deploy and run the example lootbox contract" number={7}>

Now that your example is configured, you can test, deploy, and run the example. Then, switch accounts to act as a user who received the lootbox and receive the example rewards.

Run the `npx hardhat run` command and replace `<network>` with the network that you want to deploy to. The network must be configured in `hardhat.config.ts`.

```
bash
npx hardhat run scripts/deploy.ts --network <network>
```

The deploy script also completes the following actions:

1. Approve each type of reward configured in `scripts/data/tokens.json` for the deployed contract address.

This is a mandatory step because the contract must store the assets in its own balance. It is important to ensure the deployer account owns all the assets you want to distribute as rewards.

1. Generate a Merkle tree for the allowlist.
1. Create and fund a VRF subscription if one is not provided.

```
{/ prettier-ignore /}
<Aside type="note">
```

Make sure the deployer account has enough LINK to fund the subscription. The initial funding amount is configured in `network-config.ts`. For testnets, you can use the LINK faucet.

Estimate the minimum subscription balance that VRF requires to process your randomness request. The minimum subscription balance provides a buffer against gas volatility, and only the actual cost of your request will be deducted from your account.

If your subscription is underfunded, your VRF request will be pending for 24 hours. If this happens, check the Subscription Manager to see the additional balance needed.

```
</Aside>
```

1. Add the deployed contract address as a consumer to the VRF subscription.

If you provided a subscription ID, make sure the deployer account is the owner of the subscription. Otherwise, comment out the `addVrfConsumer` function in the deploy script and add the contract address manually.

1. Verify the contract on Etherscan. This is important to show users the source code for the contract so they can confirm how it works. If you want to skip this step during testing, comment out the `verify` function in the deploy script.

Next, switch accounts and open the lootbox as a user to test the contract.

</Accordion>

<Accordion title="Open the lootbox and claim rewards" number={8}>

Once the contract is deployed and the start timestamp is reached, users can start opening the lootbox. The amount of rewards users receive depends on the amount of units they open by specifying the `amountToOpen` parameter and paying the corresponding fee.

1. Switch to your receiving account to act as a test user and copy its address for the next steps.

1. Open the lootbox. In Sepolia Etherscan:

- Call the `publicOpen()` function if you initialized the contract with an empty allowlist.
- Call the `privateOpen()` function if you set up an allowlist. Generate the Merkle proof locally using `merkletreejs`, and pass in the opener address:

```
js
merkleTree.getHexProof(keccak256(allowlistedUserAddress))
```

Pass the Merkle proof as input to `privateOpen()` in Sepolia Etherscan.

1. The rewards are ready to claim when the randomness result from VRF is ready. To check this, navigate to the Read tab, call the `canClaimRewards` function and pass in the opener address.

1. When the rewards are ready, navigate to the Write tab, call the `claimRewards` function to claim the reward and pass in the opener address.

</Accordion>

Reference

Access control

You can control access to the lootbox with two modes: private and public. Use these modes to determine which users can open the lootbox. The owner account can update this setting after deployment, so you can test the opening functionality privately before making the lootbox available to the public.

Private mode

In this mode, the lootbox is only open to addresses on the allowlist by calling the `privateOpen` function and providing Merkle proof for the address. You can generate the Merkle proof locally using `merkletreejs`:

```
js
merkleTree.getHexProof(keccak256(whitelistedUserAddress))
```

The allowlist is set on contract deployment and can be changed by calling the `setWhitelistRoot` function from the owner account. The private mode can be enabled or disabled at any time by calling the `setPrivateOpen` function from the owner account.

Public mode

When the private mode is disabled, anyone can open the lootbox by calling the `publicOpen` function. It will do the same thing as the `privateOpen` function but without the Merkle proof.

Randomness

The contract uses Chainlink VRF to generate randomness which is used to determine the rewards the user receives.

Because the randomness is generated offchain, the contract will not be able to transfer the rewards immediately. Instead, it will store the request and once the randomness is received, the user or anyone else can call the `claimRewards` function to transfer the rewards to the user.

The lootbox creator can also call the `claimRewards` function and improve the user experience by transferring the rewards to the user immediately. You can automate this further by using Chainlink Automation.

Claim rewards

The rewards for an open request can be claimed by calling the `claimRewards` function and passing the opener address as the parameter. This will transfer the rewards to the opener address.

The claim function can only be called after the randomness is fulfilled. To check this, call the `canClaimRewards` function and pass the opener address as the parameter.

One address can only have one open request at a time. If you try to open the lootbox again before the previous request is fulfilled, the transaction reverts with a `PendingOpenRequest` error.

Withdraw funds

At any time, the owner can withdraw funds from the collected fees by calling the `withdraw` function. By doing so, the contract balance will be transferred to the owner account.

Code walkthrough

The main part of this tutorial is in the `Lootbox.sol` contract. The `Lootbox` contract manages the following tasks:

- Requests a random value from VRF for each open request
- Uses the random value as a seed to determine the reward
- Tracks and validates requests to open the lootbox
- Transfers rewards to the user, for valid requests
- Tracks how many rewards remain in the lootbox
- Validates user addresses against the allowlist, if in private mode

This section focuses on how VRF is used in the `lootbox` contract.

How the lootbox interacts with the VRF service

The processing done in the `fulfillRandomWords` callback function is minimal. Here, the function simply stores the randomness result from VRF and emits an event to indicate that the randomness is ready, so the `lootbox` open request can now be fulfilled. Reducing the computational processing in your `fulfillRandomWords` callback function makes the callback more gas efficient, and it reduces the chance that your `callbackGasLimit` will be too low for the VRF request to go through.

solidity

```
...
/// @notice Requests randomness from Chainlink VRF
/// @dev The VRF subscription must be active and sufficient LINK must be
available
/// @return requestId The ID of the request
function requestRandomness() internal returns (uint256 requestId) {
```

```

        requestId = VRFCoordinatorV2Interface(ivrFCoordinatorV2)
            .requestRandomWords(
                ivrfKeyHash,
                ivrfSubscriptionId,
                REQUESTCONFIRMATIONS,
                CALLBACKGASLIMIT,
                NUMWORDS
            );
    }

    /// @inheritdoc VRFConsumerBaseV2
    function fulfillRandomWords(
        uint256 requestId,
        uint256[] memory randomWords
    ) internal override {
        srequests[requestId].randomness = randomWords[0];
        emit OpenRequestFulfilled(requestId, randomWords[0]);
    }

```

vrf-mystery-box.mdx:

```

---
title: "VRF-Enabled Mystery Box"
description: "Build a mystery box smart contract using VRF."
image: "QuickStarts-VRF-Enabled-LootBox-Pack-Contract.webp"
products: ["vrf"]
time: "90 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---

```

import { Accordion, Aside, CodeSample } from "@components"

Overview

This is a template for NFT collection with a mystery box mechanic powered by Chainlink VRF V2.5.

Smart contracts are based on the gas efficient ERC721Psi. It's super easy to deploy and configure with most of the steps automated in the deploy script.

The key features of this template include:

- a private minting stage with a merkle tree
- rate limited batch minting
- delayed reveal with randomization technique to save gas
- provenance hash to verify the authenticity of the metadata
- royalties for secondary sales
- configurable parameters

Objective

You will use a Hardhat project to deploy an existing MysteryBox application on Ethereum Sepolia.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how

to interact with Chainlink's systems and services so that you can integrate them into your own. This template is

provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key

checks or error handling to make the usage of the product more clear. Do not use the code in this example in a

production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

<Aside type="note" title="New to smart contract development?">
If you are new to smart contract development, learn how to Deploy Your First Smart Contract.
</Aside>

Before you start this tutorial, complete the following items:

- Set up a cryptocurrency wallet such as MetaMask.
- Fund your wallet with testnet ETH and LINK from faucets.chain.link/sepolia.
- Install git. Run `git --version` to check the installation.
- Install Nodejs version 16 or later.
- Optionally, you can install Yarn and use it to run this example instead of using npm.
- Create an Etherscan API key if you do not already have one. This is used to verify your contracts onchain.
- Create an account with Infura or Alchemy to obtain an RPC endpoint if you do not already have one.

Steps to implement

<Accordion title="Clone the example repo and install dependencies" number={1}>

Clone the repo and install all dependencies.

If you want to use npm, run:

```
bash
git clone https://github.com/smartcontractkit/quickstarts-mysterybox.git && \
cd quickstarts-mysterybox && \
npm install
```

Alternatively, you can use yarn to install dependencies:

```
bash
git clone git@github.com:smartcontractkit/quickstarts-mysterybox.git && \
cd quickstarts-mysterybox && \
yarn install
```

</Accordion>

<Accordion title="Configure your project" number={2}>

Copy the `.env.example` file to `.env` and fill in the values.

```
bash
cp .env.example .env
```

Set the parameters for the NFT contract. If you don't have a VRF subscription, delete the `VRFSUBSCRIPTIONID` parameter.

Parameter	Description
Example	

NFTNAME	The name of the NFT collection
MysteryBox	
NFTSYMBOL	The symbol of the NFT collection
BOX	
NFTUNREVEALEDURI	The metadata URI for all tokens before reveal
https://example.com	
NFTMAXSUPPLY	The maximum number of tokens that can be minted
100	
NFTMAXMINTPERUSER	The maximum number of tokens that can be minted per user
address	10
NFTFEE	The fee for minting a token in ETH
0.01	
NFTROYALTYBPS	The royalty fee for selling a token in basis points
500	
VRFSUBSCRIPTIONID	A funded Chainlink VRF V2.5 subscription ID. If you
leave this blank, a new subscription will be created and funded on deploy.	
79850349243438349975305816782035019118399435445660033947721688676378382535454	

</Accordion>

<Accordion title="Configure the Hardhat project" number={3}>

Hardhat is an Ethereum development environment that is used here to configure and deploy the mystery box contract. You need the following information:

Parameter	Description
Example	

SEPOLIARPCURL	The RPC URL for the Ethereum Sepolia network.
https://eth-sepolia.g.alchemy.com/v2/...	
PRIVATEKEY	The private key of the account you want to deploy from.
abc123abc123abc123abc123abc123...	
SCANNERAPIKEY	The API key for Etherscan used for contract verification.
ABC123ABC123ABC123ABC123ABC123ABC1	

</Accordion>

<Accordion title="Test the contracts locally" number={4}>

To run the unit tests, run the following command:

```
bash
npm run test
```

If you want to see gas usage, run the following command:

```
bash
REPORTGAS=true npm run test
```

For coverage reports, run the following command:

```
bash
```

```
npm run coverage
```

</Accordion>

<Accordion title="Deploy the example contract" number={5}>

Run the `npx hardhat run` command and replace `<network>` with the network that you want to deploy to. The network must be configured in `hardhat.config.ts`.

```
bash
npx hardhat run scripts/deploy.ts --network <network>
```

In addition to deploying the contract, the deploy script also completes the following steps automatically:

1. Create and fund a VRF V2.5 subscription if one is not provided. Make sure the deployer account has enough LINK to fund the subscription. The initial funding amount is configured in `network-config.ts`. For testnets, you can use the LINK faucet to get LINK. If your subscription is underfunded, your VRF request will be pending for 24 hours. If this happens, check the Subscription Manager to see the additional balance needed.

1. Add the deployed contract address as a consumer to the VRF subscription. If you provided a subscription ID, make sure the deployer account is the owner of the subscription. Otherwise, comment out the `addConsumerToSubscription` function in the deploy script and add the contract address manually.

1. Generate a Merkle tree for the allowlist. The merkle tree is generated from the address list in `scripts/data/whitelist.json` file.

1. Verify the contract on Etherscan. This is important to show users the source code for the contract so they can confirm how it works. It's also required to run the example yourself.

After the deployment is complete, the terminal prints the link to view your contract on Etherscan.

</Accordion>

<Accordion title="Run the example" number={6}>

After the contract is deployed, run the example using contract functions on Etherscan.

1. Open the Etherscan link from your terminal to view your contract.
1. On the Contract tab, click the Write Contract tab to view your contract's write functions.
1. Click Connect to Web3 to connect your wallet to Etherscan so you can run the functions.
1. Enable public minting:
1. Open the `setPublicMint` function.
1. Enter a value of `true`.
1. Click Write to run the function. Approve the transaction in MetaMask.
1. Mint an NFT to test the contract:
1. Open the `publicMint` function.
1. Enter a `publicMint` value of `0.01 ETH` and an amount of `1`.
1. Click Write to run the function. Approve the transaction in MetaMask.
1. Reveal the box:
1. Open the `reveal` function.
1. Click reveal to reveal the mystery box.
1. After the reveal, you can find the NFT in the token list for your externally owned account (EOA).

</Accordion>

Explore the code

After you complete the example, see the Mystery Box repository to learn about more capabilities of this example and how to configure them. For example, you can learn more about the various Configuration Options for this example and modify it to meet the needs of your application.

```
# vrf-subscription-monitor.mdx:
```

```
---
title: "VRF Subscription Balance Monitor"
description: "Automatically top-up your VRF subscription balances using Chainlink Automation to ensure there is sufficient funding for requests."
image: "QuickStarts-VRF-Balance-Subscription-Manager.webp"
products: ["automation", "vrf"]
time: "30 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---
```

```
import { Accordion, Address, Aside, ClickToZoom, CodeSample, CopyText } from
"@components"
```

Overview

Automatically top-up your VRF subscription balances using Chainlink Automation to ensure there is sufficient funding for requests. View the code on GitHub.

Objective

This example shows how to automate the process for funding VRF subscription balances. You will deploy and test a VRF subscription balance manager contract that monitors multiple subscriptions and tops them up with LINK as necessary. You can set up this contract to monitor existing VRF subscriptions. Alternatively, you will create two VRF subscriptions for testing: one that is underfunded, and another that is adequately funded. When you test the subscription balance manager contract, it only tops up the underfunded subscription.

<Aside type="caution" title="Disclaimer">

This tutorial represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

</Aside>

Before you begin

Before you start this tutorial, complete the following items:

- If you are new to smart contract development, learn how to Deploy Your First Smart Contract.

- Install and configure a cryptocurrency wallet like MetaMask.
- Get Sepolia testnet LINK and ETH from faucets.chain.link.
- Gather the subscription IDs for any existing VRF subscriptions you would like to monitor. If you do not have any VRF subscriptions, you will create two for demo purposes in this tutorial.

<Aside type="note" title="New to smart contracts?">

This tutorial assumes that you know how to create and deploy basic smart contracts. If you are new to smart contract development, deploy a VRF-compatible contract by following these instructions and then return to this tutorial.

</Aside>

Steps to implement

This tutorial requires you to set up the following components in VRF before you use Automation to monitor VRF subscriptions:

- Create VRF subscriptions to monitor, if you don't already have any
- Deploy VRF-compatible contracts

If you already have existing VRF subscriptions you would like to monitor, skip to the next step.

<Accordion title="Create VRF subscriptions to monitor" number={1}>

If you don't already have existing VRF subscriptions you would like to monitor, create two VRF subscriptions for demo purposes. One subscription will be adequately funded, and the other subscription will be underfunded intentionally so that the subscription balance monitor contract can fund it.

1. Open the VRF Subscription Manager, and connect your wallet.

```
{/ prettier-ignore /}
{" "}
```

```
<div class="remix-callout">
  <a href="https://vrf.chain.link">Open the Subscription Manager</a>
</div>
```

1. Create two subscriptions:

- Fund one subscription with 12 testnet LINK to serve as the adequately funded VRF subscription
- Fund the other subscription with 1 testnet LINK to serve as the underfunded VRF subscription

</Accordion>

<Accordion title="Deploy VRF-compatible contracts as consumers" number={2}>

In this section, you will deploy the same VRF contract twice. Each deployed contract will serve as a consuming contract for one of your VRF subscriptions.

1. Deploy this VRF-compatible contract for your VRF subscription:

```
<CodeSample src="/samples/VRF/VRFD20.sol" showButtonOnly />
```

This VRFD20.sol contract uses VRF to randomly assign you a Game of Thrones house. The VRF coordinator address and key hash values are hardcoded in this contract for Sepolia. If you want to use a different testnet, you must update those values before deploying the contract.

<Aside type="note">

If you are unfamiliar with deploying smart contracts, deploy a VRF-compatible contract by following the VRF

Getting Started guide. Then return to this tutorial.

</Aside>

1. Under the Solidity compiler tab, compile the contract.

1. Under the Deploy and run transactions tab, select Injected Provider - MetaMask for the Environment field. Make sure the VRFD20.sol contract is selected in the Contract field.

1. Deploy the contract, passing in the subscription ID of your adequately funded subscription.

1. After the contract is deployed successfully, copy the contract address from Remix. Navigate back to the VRF Subscription Manager and add the contract address as a consumer for your adequately funded subscription.

1. Navigate back to Remix to deploy the contract once more. Change the subscription ID to the subscription ID of your intentionally underfunded subscription, and click Deploy.

1. Once more, after the contract is deployed successfully, copy the second contract address from Remix. Navigate back to the VRF Subscription Manager and add this second contract address as a consumer for your intentionally underfunded subscription.

</Accordion>

<Accordion title="Deploy the subscription balance monitor contract" number={3}>

In this section, you will deploy the subscription balance monitor contract, using the same address that is the admin owner for both of your VRF subscriptions. The contract will monitor any VRF subscriptions owned by this address, and fund any underfunded subscriptions using funds owned by this address.

Get required inputs for the balance monitor contract

The constructor for this contract requires the following information for the supported network that you want to deploy on:

- The LINK token address, which is found on the LINK token contracts page.
- The Automation registry address, which is found on the Automation supported networks page.
- The VRF coordinator address, which is found on the VRF Subscription Supported Networks page.
- Minimum wait period, which you can use to add buffer time between funding multiple subscription IDs. For demo purposes, you will monitor only two VRF subscriptions, so you can set this value to 60 seconds.

For Sepolia, these values are all consolidated here:

Item	Value

LINK token address	<Address contractUrl="https://sepolia.etherscan.io/token/0x779877A7B0D9E8603169DdbD7836e478b4624789" urlId="111551110x779877A7B0D9E8603169DdbD7836e478b4624789" urlClass="erc-token-address"/>
VRF Coordinator	<Address

```
contractUrl="https://sepolia.etherscan.io/address/0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625" />
|
```

1. Open the VRFSubscriptionBalanceMonitor.sol contract in Remix.

```
<CodeSample
src="/samples/Automation/tutorials/VRFSubscriptionBalanceMonitor.sol"
showButtonOnly />
```

1. Under the Solidity compiler tab, compile the contract.

1. Under the Deploy and run transactions tab, select Injected Provider - MetaMask for the Environment field.

1. Expand the Deploy field and enter the values for Sepolia:

```
| Item | Value
| ----- |
| LINKTOKENADDRESS | <CopyText
text="0x779877A7B0D9E8603169DdbD7836e478b4624789" code/> |
| COORDINATORADDRESS | <CopyText
text="0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625" code/> |
| MINWAITPERIODSECONDS | <CopyText text="60" code/>
|
```

1. Click Deploy. MetaMask opens and prompts you to confirm the contract deployment transaction.

1. Copy the address of your newly deployed contract. You can find this in Sepolia Etherscan through the contract deployment transaction, or in Remix in the Deployed Contracts section.

</Accordion>

<Accordion title="Register the subscription balance monitor contract on Automation" number={4}>

In this section, you will register an upkeep on Chainlink Automation to run the subscription balance monitor contract. This enables Automation to check whether you have any underfunded VRF subscriptions, and if so, top up their balances appropriately.

Register an upkeep

Registering an upkeep on Chainlink Automation creates a smart contract that will run your VRF subscription balance monitor contract.

```
{/ prettier-ignore /}
{" "}
```

```
<div class="remix-callout">
  <a href="https://automation.chain.link">Open the Chainlink Automation App</a>
</div>
```

1. Click Register new Upkeep.

1. Select the Custom logic trigger option.

1. Copy your contract address from Remix and paste it into the Target contract address field.

1. Specify a name for your upkeep and set a Starting balance of 5 LINK for this demo. Leave the other settings at their default values.

1. Go to the Upkeep details page and copy your Forwarder address.

1. Navigate back to Remix and find the setForwarderAddress function. Paste your forwarder address and click click transact to run the function. MetaMask asks you to confirm the transaction.

Now that you've registered the upkeep and configured your contract with your new upkeep's forwarder address, Chainlink Automation handles the rest of the process.

</Accordion>

<Accordion title="Configure the subscription watch list" number={5}>

Before your contract will fund a subscription, you must set the watchList address array with minBalancesJuels and topUpAmountsJuels variables. For demonstration purposes, you configure your own wallet as the top-up address. This makes it easy to see the ETH being sent as part of the automated top-up function. After you complete this tutorial, you can configure any wallet or contract address that you want to keep funded.

1. In the list of functions for your deployed contract, run the setWatchList function. This function requires an subscriptionIds array, a minBalancesJuels array that maps to the subscriptions, and a topUpAmountsJuels array that also maps to the subscriptions. In Remix, arrays require brackets and quotes around integer values. For this example, set the following values:

- subscriptionIds: ["SUBID1", "SUBID2"]
- minBalancesJuels: ["2000000000000000000", "2000000000000000000"]
- topUpAmountsJuels: ["100000000000000000", "100000000000000000"]

These values tell the top up contract to top up the specified address with 0.01 LINK if the address balance is less than 2 LINK. These settings are intended to demonstrate the example using testnet faucet funds. For a production application, you might set more reasonable values that top up a smart contract with 10 LINK if the balance is less than 1 LINK.

1. After you configure the function values, click transact to run the function. MetaMask asks you to confirm the transaction.

1. In the functions list, click the getWatchList function to confirm your settings are correct.

</Accordion>

Examine the code

This section explains how the VRFSubscriptionBalanceMonitor.sol contract tracks your VRF subscriptions and uses Chainlink Automation to fund your underfunded subscriptions. You can view the code for the full contract on GitHub.

Tracking subscriptions

The VRFSubscriptionBalanceMonitor.sol contract tracks your VRF subscriptions by creating a watchlist and storing attributes of each subscription required to monitor when they need funding.

The setWatchList() function creates the watchlist of subscriptions, validates it, and then sets a Target for each subscription. The Target struct helps track whether a subscription is active, its minimum balance, when it was last funded, and the amount of LINK (in juels) to send to the subscription each time it is

topped up.

```
solidity
struct Target {
    bool isActive;
    uint96 minBalanceJuels;
    uint96 topUpAmountJuels;
    uint56 lastTopUpTimestamp;
}

...

/
    @notice Sets the list of subscriptions to watch and their funding
parameters.
    @param subscriptionIds the list of subscription ids to watch
    @param minBalancesJuels the minimum balances for each subscription
    @param topUpAmountsJuels the amount to top up each subscription
/
function setWatchList(
    uint64[] calldata subscriptionIds,
    uint96[] calldata minBalancesJuels,
    uint96[] calldata topUpAmountsJuels
) external onlyOwner {
    if (subscriptionIds.length != minBalancesJuels.length ||
subscriptionIds.length != topUpAmountsJuels.length) {
        revert InvalidWatchList();
    }
    uint64[] memory oldWatchList = swatchList;
    for (uint256 idx = 0; idx < oldWatchList.length; idx++) {
        targets[oldWatchList[idx]].isActive = false;
    }
    for (uint256 idx = 0; idx < subscriptionIds.length; idx++) {
        if (targets[subscriptionIds[idx]].isActive) {
            revert DuplicateSubscriptionId(subscriptionIds[idx]);
        }
        if (subscriptionIds[idx] == 0) {
            revert InvalidWatchList();
        }
        if (topUpAmountsJuels[idx] <= minBalancesJuels[idx]) {
            revert InvalidWatchList();
        }
        targets[subscriptionIds[idx]] = Target({
            isActive: true,
            minBalanceJuels: minBalancesJuels[idx],
            topUpAmountJuels: topUpAmountsJuels[idx],
            lastTopUpTimestamp: 0
        });
    }
    swatchList = subscriptionIds;
}
```

The `getUnderfundedSubscriptions()` function assesses each subscription in the watchlist and creates a list of subscriptions that need to be funded. Using the attributes stored in the `Target` struct for each subscription, this function adds a subscription to the `needsFunding` list if the following conditions are met:

- The subscription is underfunded
- The owning contract has a sufficient balance to fund the subscription
- It's been long enough since the last time the subscription's balance was topped up


```

solidity
/
  @notice Gets a list of subscriptions that are underfunded.
  @return list of subscriptions that are underfunded
/
function getUnderfundedSubscriptions() public view returns (uint64[] memory) {
    uint64[] memory watchList = swatchList;
    uint64[] memory needsFunding = new uint64[](watchList.length);
    uint256 count = 0;
    uint256 minWaitPeriod = sminWaitPeriodSeconds;
    uint256 contractBalance = LINKTOKEN.balanceOf(address(this));
    Target memory target;
    for (uint256 idx = 0; idx < watchList.length; idx++) {
        target = targets[watchList[idx]];
        (uint96 subscriptionBalance, , , ) =
COORDINATOR.getSubscription(watchList[idx]);
        if (
            target.lastTopUpTimestamp + minWaitPeriod <= block.timestamp &&
            contractBalance >= target.topUpAmountJuels &&
            subscriptionBalance < target.minBalanceJuels
        ) {
            needsFunding[count] = watchList[idx];
            count++;
            contractBalance -= target.topUpAmountJuels;
        }
    }
    if (count < watchList.length) {
        assembly {
            mstore(needsFunding, count)
        }
    }
    return needsFunding;
}

```

Using Automation

The VRFSubscriptionBalanceMonitor.sol contract uses Chainlink Automation to top up underfunded subscriptions:

- In checkUpkeep(), it pulls the list of underfunded subscriptions. If this list is empty, it indicates that the upkeep is not needed, and Automation takes no further action.
- Automation only runs performUpkeep() if there are underfunded subscriptions, and it tops up any subscriptions that need funding.

```

solidity
/
  @notice Gets list of subscription ids that are underfunded and returns a
keeper-compatible payload.
  @return upkeepNeeded signals if upkeep is needed, performData is an abi
encoded list of subscription ids that need funds
/
function checkUpkeep(
    bytes calldata
) external view override whenNotPaused returns (bool upkeepNeeded, bytes memory
performData) {
    uint64[] memory needsFunding = getUnderfundedSubscriptions();
    upkeepNeeded = needsFunding.length > 0;
    performData = abi.encode(needsFunding);
    return (upkeepNeeded, performData);
}
/

```

```

    @notice Called by the keeper to send funds to underfunded addresses.
    @param performData the abi encoded list of addresses to fund
  /
function performUpkeep(bytes calldata performData) external override
onlyKeeperRegistry whenNotPaused {
    uint64[] memory needsFunding = abi.decode(performData, (uint64[]));
    topUp(needsFunding);
}

```

web3js.mdx:

```

---
title: "Using the Web3.js Plugin"
description: "Incorporate the web3.js plugin to your NodeJS project with a
single line of code to natively interact with Chainlink Data Feeds."
image: "QuickStarts-web3.js-plugin.webp"
products: ["general"]
time: "30 minutes"
requires: "Wallet with gas token & ERC-677 LINK"
---

```

```
import { Accordion } from "@components"
```

Objective

Learn to use the web3.js ^4.0.0 plugin for interacting with Chainlink Data Feeds. This tutorial shows you how to use a Price Feed in a project that already uses web3.js. The plugin simplifies the process so you don't have to handle all of the details manually.

Before you begin

- Install NodeJS
- Install Yarn

Steps to implement

```
<Accordion title="Integrate the plugin with your existing project" number={1}>
```

Use the following steps to integrate the @chainsafe/web3-plugin-chainlink with your existing project.

1. Clone your current project repository or create a new web3.js project.

1. From your current project directory, install the @chainsafe/web3-plugin-chainlink.

```

shell
yarn add @chainsafe/web3-plugin-chainlink

```

1. Install web3version ^4.0.0 or later in your project. To verify you have the correct web3 version installed, look at the versions listed in your project's package.json under the dependencies section for "web3": "4.1.1" or a similar version.

```

shell
yarn add web3@4.1.1

```

```
</Accordion>
```

<Accordion title="Register the plugin with a web3.js instance" number={2}>

After importing ChainlinkPlugin from @chainsafe/web3-plugin-chainlink and Web3 from web3, register an instance of ChainlinkPlugin with an instance of Web3 like the following example:

```
typescript
import { ChainlinkPlugin } from "@chainsafe/web3-plugin-chainlink"
import { Web3 } from "web3"

const web3 = new Web3("YOURPROVIDERURL")
const chainlinkPlugin = new ChainlinkPlugin()

web3.registerPlugin(chainlinkPlugin)
```

</Accordion>

<Accordion title="Run the tests" number={3}>

Run yarn to install dependencies. If you receive the following warning, remove the file package-lock.json and make sure to run yarn to install dependencies instead of npm i:

```
shell
warning package-lock.json found. Your project contains lock files generated by
tools other than Yarn. It is advised not to mix package managers in order to
avoid resolution inconsistencies caused by unsynchronized lock files. To clear
this warning, remove package-lock.json.
```

Run the following tests:

- yarn test:unit: Runs the mocked tests that do not make a network request using the Jest framework
- End-to-end tests: Runs Webpack bundled tests that make a network request to the RPC provider https://rpc.ankr.com/eth and returns an actual response from MainnetPriceFeeds.LinkEth smart contract using the Cypress framework
 - yarn test:e2e:chrome: Runs the tests using Chrome
 - yarn test:e2e:electron: Runs the tests using Electron
 - yarn test:e2e:firefox: Runs the tests using Firefox
- Black box tests: Uses a published version of the plugin from Verdaccio to run tests that make a network request to the RPC provider https://rpc.ankr.com/eth and returns an actual response from MainnetPriceFeeds.LinkEth smart contract using the Jest framework
 - Note that the black box tests are setup to run within Github actions environment, but can be ran locally. The blackboxtesthelpers.sh script can be used to:
 - start: Start Verdaccio using a Docker container
 - stop: Kill the Docker container
 - startBackgroundAndPublish: Starts a headless Docker container and publishes the plugin package
 - runTests: cds into the test/blackbox directory, installs the black box package dependencies, and runs yarn test which will use Jest to run the tests
 - In addition to the blackboxtesthelpers.sh script, the black box tests can be ran using the following package.json scripts:
 - yarn pre-black-box: Calls startBackgroundAndPublish from the blackboxtesthelpers.sh script
 - yarn test:black-box: Calls yarn pre-black-box and runTests from the from the blackboxtesthelpers.sh script
 - yarn post-black-box: Calls stop from the blackboxtesthelpers.sh script

More information about registering web3.js plugins can be found [here](#).

</Accordion>

<Accordion title="Using Price Feed Addresses" number={4}>

Included in this plugin are two enums that contain the Ethereum contract addresses for specific token pairs:

- MainnetPriceFeeds
- GoerliPriceFeeds

If you cannot find your desired price feed within these enums, check the Price Feed Addresses page to make sure its supported. If it is, open an issue or a pull request for the missing price feed so that it can be added to the appropriate enum.

Add the getPrice function to your code.

```
typescript
async getPrice(
    priceFeedAddress: MainnetPriceFeeds | GoerliPriceFeeds | Address,
    aggregatorInterfaceAbi: ContractAbi = defaultAggregatorInterfaceAbi,
): {
    roundId: bigint,
    answer: bigint,
    startedAt: bigint,
    updatedAt: bigint,
    answeredInRound: bigint
}
```

The defaultAggregatorInterfaceAbi can be found in the GitHub repository.

The getPrice method, accepts MainnetPriceFeeds | GoerliPriceFeeds | Address for its first parameter, and an optional second parameter for specifying the Chainlink Aggregator Interface ABI of the Ethereum smart contract you'd like to interact with (the parameter is defaulted to defaultAggregatorInterfaceAbi).

Under the hood, this method is calling the latestRoundData for the specified price feed. For more information, see the Data Feeds API Reference.

Your complete code should look similar to the following example:

```
typescript
import { ChainlinkPlugin, MainnetPriceFeeds } from "@chainsafe/web3-plugin-chainlink"
import { Web3 } from "web3"

const web3 = new Web3("YOURPROVIDERURL")
const chainlinkPlugin = new ChainlinkPlugin()

web3.registerPlugin(chainlinkPlugin)

web3.chainlink.getPrice(MainnetPriceFeeds.LinkEth).then(console.log)
// {
//   roundId: 73786976294838212867n,
//   answer: 4185000000000000n,
//   startedAt: 1674178043n,
//   updatedAt: 1674178043n,
//   answeredInRound: 73786976294838212867n
// }
```

</Accordion>

```
# acquire-link.mdx:
```

```
---
section: global
date: Last Modified
title: "Acquire testnet LINK"
whatsnext: { "Deploy your first contract": "/quickstarts/deploy-your-first-
contract" }
---
```

The Getting Started guides show you how to send ETH on the Sepolia testnet, but some contracts might require you to use LINK token instead. This page shows you how to obtain testnet LINK and send it to your MetaMask wallet.

Configure MetaMask to use LINK tokens

To see your LINK token balance in MetaMask, you must manually add the token.

1. Open up MetaMask.
1. At the bottom of the MetaMask windows, click Import tokens.
1. Find the LINK token contract address for the network that you want to use. On Sepolia, the LINK token address is: 0x779877A7B0D9E8603169DdbD7836e478b4624789. See the LINK Token Contracts page to find the addresses for different testnets.
1. Paste the token contract address into MetaMask in the Token Address input. The token symbol and decimals of precision will auto-populate.

!Metamask Custom Tokens Screen

1. Click Next. A new window will appear, showing the LINK token details.
1. Click Import Tokens to confirm adding the new token.

MetaMask should now display the new LINK token balance.

Get testnet LINK from a faucet

1. Go to faucets.chain.link.
1. In MetaMask, select the network where you want to receive testnet LINK.
1. Click Connect wallet so the faucet app can detect the network and wallet address.
1. If you want to receive testnet funds at a different address, paste it in the Wallet address section. This field defaults to your connected wallet address.
1. In the Request type section, select the testnet funds that you want to receive.
1. Complete the Captcha and click Send request. The funds are transferred from the faucet to the wallet address that you specified.

After the transaction is confirmed onchain, the faucet app shows "Request complete" and the transaction hash of your request.

!Successful Faucet Request Message

```
# bridge-risks.mdx:
```

```
---
section: global
date: Last Modified
title: "Cross-chain bridges and associated risks"
---
```

```
import { Aside } from "@components"
```

When working with Chainlink on layer-2 chains and sidechains, you must export your LINK tokens from Ethereum to the target chain using a cross-chain bridge.

Follow this video for an example of moving LINK tokens from Ethereum to Polygon.

Cross-chain bridges come with their own risks. In fact, bridge attacks constitute some of the largest cryptocurrency hacks by value. When moving your LINK tokens or any asset across chains, understand the risks that you are taking with your assets. Chainlink Labs does not endorse any bridge. Ultimately, you are responsible for assessing the bridge that you use to move your assets.

Read the What is a cross-chain bridge, Trade-offs, and Risks sections to learn more about bridges and trust assumptions in their designs. After you read these sections, you will have a better understanding of bridge risks and which aspects you should evaluate when using a bridge.

What is a Cross-chain bridge

With the proliferation of layer-1 blockchains and layer-2 scaling solutions, the web3 ecosystem has become multi-chain. Each blockchain comes with its own approach to scalability, security, and trust.

However, blockchains are not natively capable of communicating with each other, which makes blockchain interoperability protocols critical for allowing dApps to interact with any onchain network and tap into each blockchain's unique assets and features.

A bridge is a core element of cross-chain interoperability. Bridges exist to connect blockchain networks and enable connectivity between them.

Bridges enable the following:

- Cross-chain transfer of assets and information
- dApps can leverage the strengths and benefits of different chains
- Collaboration between developers from different blockchain ecosystems to build new platforms and products for users

As an analogy, you can use the blockchains as cities mental model:

- Layer-1 blockchains are like cities.
- Layer-2 solutions are equivalent to skyscrapers. As described in the mental model, "Each rollup is like a vertical blockchain that extends from the ground L1".
- Bridges are like roads and streets that connect different cities and skyscrapers.

Trade-offs

With the growing number of layer-1 and layer-2 chains, the number of bridges has also grown, surpassing one hundred. So, how do you choose the correct bridge?

When choosing a bridge, there is no perfect solution, only trade-offs. As explained in the interoperability trilemma and Ethereum foundation docs, bridge designs must compromise between the following characteristics:

- Trust-minimization: The system does not introduce new trust or security assumptions beyond those of the underlying blockchains. Read trust-minimization for more details.
- Generalizability: The system enables the transfer of complex arbitrary data. Data could be messages or assets/funds.
- Extensibility: How hard is it to integrate a new blockchain?
- Latency: How long does it take to complete a transaction?
- Costs: How much does it cost to transfer data across chains via a bridge?

Risks

When choosing a bridge, be aware of the following risks.

Smart contract risks

Bugs and vulnerabilities can expose users' assets to different kinds of exploits. Read this detailed analysis for an example of a bridge exploit where the attacker could leverage a logical error in the bridge's smart contract.

Systemic financial risks

To transfer tokens cross-chain, many bridges lock tokens on the source chain and mint derivative or wrapped tokens on the destination chain representing the locked tokens. A hack of the locked tokens or an infinite mint attack on the wrapped tokens can make all wrapped tokens worthless and expose entire blockchains to risk.

Early stage

Given that bridges are relatively new, there are many unanswered questions related to how bridges will perform in different market conditions.

Trust-minimization (Counterparty risk)

To overcome cross-chain interoperability challenges, some bridges use offchain actors or validators. These actors introduce new trust assumptions in addition to the underlying blockchain trust assumptions. These bridges act as a custodian and are, therefore, trust-based.

In contrast, some bridge designs rely on underlying blockchains' validators and, therefore, do not add any trust assumptions.

To summarize:

- Trusted (custodial) bridges require a third party to validate movements over the bridge. Users are required to give up control of their crypto assets, so trust is involved as they rely on the bridge operator's reputation.
- Trustless (non-custodial) bridges leverage smart contracts to store and release funds on either side of the bridge. These bridges are trust-minimized because they don't make new trust assumptions beyond the underlying blockchains.

Trustlessness in bridges does not exist in an absolute form (trusted vs. trustless). As explained in the blockchain-interoperability blog, there are four general interoperability solutions for validating the state of a source blockchain and relaying the subsequent transaction to the destination blockchain:

Web2 Verification

Web2 verification is when someone uses a web2 service to execute a cross-chain transaction. The most common example in practice is when users leverage centralized exchanges to swap or bridge their own tokens. The user simply deposits their assets into an address on the source chain that's under the control of the exchange and then withdraws the same tokens or different tokens (via a swap on the exchange) to an address on a destination chain controlled by the user.

Web2 verification can be fairly convenient for personal transactions and requires less technical expertise. However, it is limited only to swapping and bridging tokens which requires trust in a centralized custodian.

External verification

External verification is where a group of validator nodes are responsible for verifying transactions. These validators do not belong to either of the two blockchains' validator sets and they also have their trust assumptions irrespective of the underlying blockchains.

External verification typically requires an honest majority assumption, where a majority of the external validator nodes must behave honestly for the integrity of the cross-chain interaction to be upheld. However, additional techniques can be used to increase trust-minimization, such as:

- Optimistic bridge verification
- Risk management networks
- Cryptoeconomic staking

Despite an additional trust assumption, external verification is currently the only practical way to perform cross-chain contract calls between certain types of blockchains while still providing trust-minimized guarantees. It's also a highly generalized and extensible form of cross-chain computation that is capable of supporting more complex cross-chain applications.

<Aside type="note" title="Cryptography risk">

Some externally verified bridges are secured by multisig wallets. Ronin is one example. Multisig wallets are also referred to as m-of-n multisigs, with M being the required number of signatures or keys and N being the total number of signatures or keys (m-of-n). This means that an attacker only needs to exploit M keys to be able to hack the whole system. In this case, users must trust that the third party is decentralized enough, signers are independent of each other, and that each signer has proper key management in place. Read this detailed analysis for an example of a bridge exploit where the attacker could compromise M keys.

</Aside>

<Aside type="note" title="Optimistic bridges">

Optimistic bridges rely on honest watchers to monitor the bridges' operations and report any risks. Because the watchers of an optimistic system are permissionless, there is no way to know if there is not at least one single watcher monitoring the system. Therefore, the cost of a successful attack is limitless as it requires an attacker to know who the watchers are and hack all of them.

</Aside>

Here are some examples of externally verified bridges that use different techniques to increase trust-minimization :

- Binance bridge is a trusted bridge using the security standards of Binance.
- Polygon POS bridge uses a proof of stake (PoS) consensus algorithm for network security.
- Nomad is an optimistic bridge. It uses optimistic verification where messages are optimistically signed on the origin chain and a timeout period is enforced on the destination. During this period, a set of actors called watchers can inspect the messages and flag any detected risks.

Local verification

Local verification is when the counterparties in a cross-chain interaction verify the state of one another. If both deem the other valid, the cross-chain transaction is executed, resulting in peer-to-peer cross-chain transactions. Cross-chain swaps using local verification are often referred to as atomic swaps.

This model has a high level of trust-minimization given reasonable blockchain assumptions, as the swap either happens or both transactions fail. Furthermore, the model works so long as both parties are economically adversarial: they cannot collude to steal funds during atomic swaps.

Note that local verification is not very generalizable to a variety of cross-chain contract calls, and comes with tradeoffs like the inadvertent call option problem – a situation where the second party in an atomic swap can either act or not act on the swap, giving them an inadvertent call option for a certain period of time. Thus, local verification is mostly used in cross-chain liquidity protocols involving liquidity pools that exist independently on each chain.

Hop or Connexx Legacy are examples of locally verified bridges.

Native verification

In this design, the validators of the destination blockchains are responsible for verifying the state of the source blockchain to confirm a given transaction. This is typically done by running a light client of the source chain in the virtual machine of the destination chain or running them both side-by-side. Native verification is the most trust-minimized form of cross-chain communication, but it is more expensive, offers less development flexibility, and is more suited to blockchains with similar state machines, such as between Ethereum and EVM-based layer-2 networks or only among Cosmos SDK-based blockchains.

The NEAR Rainbow Bridge is an example of a natively verified bridge. A smart contract with Ethereum light client functionality is deployed on the NEAR blockchain and a smart contract with NEAR protocol light client functionality is deployed on Ethereum. These light clients hold the latest block headers and verify that cross-chain transactions are done across both chains. The trust model relies only on Ethereum and Near validators.

```
# contributing-to-chainlink.mdx:
```

```
---
section: global
date: Last Modified
title: "Contributing to Chainlink"
---
```

```
import { YouTube } from "@astro-community/astro-embed-youtube"
```

Chainlink is an open-source project licensed under the MIT license.

```
<YouTube id="https://www.youtube.com/watch?v=nerpcSPN4kE" />
```

What it means to contribute

When you contribute to the Chainlink project, you as a developer or community member contribute your time and effort to help improve and grow Chainlink. Your contribution can be from various methods:

- Building and maintaining the Chainlink software and tools
- Improving and maintaining the documentation, including translations into other languages
- Creating Chainlink focused content (blog posts, tutorials, videos etc)
- Becoming a developer expert
- Becoming a community advocate
- Running a Chainlink focused developer Bootcamp (in person or online)
- Running an in-person meetup or watch party
- Participate in a hackathon
- Applying for a grant

Why should you contribute?

Open source software is a model that brings multiple benefits for both the

project and the contributors. As a developer or community member, contributing to Chainlink helps you to gain valuable skills and experience, improve the software that you use, and grow your personal brand in the community which can lead to future employment opportunities. On top of these awesome things, contributing to open source is fun. It can give you a sense of community involvement, and gives you a personal sense of satisfaction knowing that you're part of an effort to build something that will enable a fairer, more transparent, and efficient new world.

Contributing to software and tooling

The most direct way you can contribute to Chainlink is to contribute to the core code or the various tooling found in our GitHub repository. Contributing to code or code-based tools can generally be split into a few different categories:

- Raising an issue
- Requesting a new feature
- Submitting a Pull Request (PR) for a fix, improvement, or new tool

Raising an issue

During the course of using Chainlink software or tools, you might encounter errors or unexpected behavior that leads you to believe the software isn't behaving correctly. You can bring this to the attention of the Chainlink Labs team as well as the wider developer community by raising an issue in the project's GitHub repository. The 'Issues' tab lists all of the open issues for the repository.

After an issue is raised and tagged, the Chainlink Labs team and the wider community can address it. This gives the issue the visibility required for someone to investigate it and resolve the issue.

When you first create an issue, you must also categorize it. This prefixes the issue name to give viewers an indication of what category the issue relates to:

- [NODE]: The issue relates to the core node software
- [DEVEL]: The issue is a result of working on code found in the current repository
- [FEAT]: The issue relates to a new feature request
- [SMRT]: The issue related to using Chainlink smart contracts
- [EXPL]: The issue related to using the Chainlink Explorer
- [FAUC]: The issue related to using the Chainlink Faucet

!Selecting the new issue category

After you select a category, enter the details for the issue. Include as much detail about the issue as possible. Provide a thorough description, environment, and software version details. Also provide detailed steps that describe how to reproduce the issue. The more thorough you make your description, the better the chances are that someone will be able to pick up the issue and resolve it.

Once a team member acknowledges that the issue has been received, they will tag it with an appropriate label. You should then monitor the state of the open issue for any questions or updates.

Requesting a new feature

Have you thought of an improvement or an awesome new feature that you think should be implemented into Chainlink? Request a new feature to bring it to the attention of the team and the wider community. You can request new features by creating a new GitHub issue in the correct repository and tagging that issue with the [FEAT] prefix (Feature request). The process for doing this is covered in the Raising an Issue section. Provide as much detail as possible in your feature request, including any benefits, risks, or considerations that you can

think of.

Voting on new features

Sometimes a new feature is put to a vote to decide if it's something that the team and wider community should implement. When a feature is put to a vote, the issue is tagged with the 'needs votes' label. You can contribute to the voting process by reacting to the first post in the feature request with a thumbs up or thumbs down emoji. This will help drive the decision. You can also contribute your thoughts by replying directly to the feature request with a new post in the thread.

!Voting on a new feature request

Submitting a pull request

The best way to contribute to Chainlink is to submit a pull request (PR). PRs can be submitted for various reasons, such as fixing an identified issue, adding a feature or improvement to the project, or even adding an entirely new repository to the Chainlink source code for a new tool or feature. If you're looking for something to pick up and create a PR for, you can search through the Chainlink repositories to find open issues, and approved feature requests.

If you're new to contributing to open-source software or Chainlink, we've tagged some good first issues against the main node software and smart contracts that you can tackle. Each major repository in the Chainlink GitHub should also have some good first issues tagged for developers to be able to take on.

All code changes must follow the [style guide] (<https://github.com/smartcontractkit/chainlink/wiki/Code-Style-Guide>), All PRs must be in an appropriately named branch with a format like 'feat/feature-description' or 'devel/issue-description'. After you submit a PR, you should get a response by a team member within a day or two acknowledging that the PR has been received. After that, monitor the PR for any additional questions or updates that come up while the team and the community review the changes.

Contributing to the documentation

The Chainlink documentation is the go-to place for developers who want to learn how to build applications using Chainlink, and node operators wanting useful information on running a Chainlink node. The documentation is open source, allowing for other developers and community members to contribute to adding or improving it. You can contribute to the Chainlink documentation in various ways:

- Improving the readability of pages
- Fixing typos or grammar errors
- Adding new guides or tutorials that you would find useful
- Translating the documentation into other languages

The process for contributing to the documentation follows the process defined earlier in the Submitting a Pull Request section. Each page also has a 'Suggest Edits' link on the top right, and will directly take you to the page in the documentation repository, where you can create a new PR with the suggested changes. Before you create a PR for the documentation, read the contributing guidelines.

If you want to translate the documentation into a new language that is not yet supported, feel free to reach out to the team beforehand, so we can make sure you get the support you need.

Creating community content

Chainlink has a strong and vibrant community of developers and community advocates. Community members often create Chainlink-focused content in various

forms and publish it for the wider community on various platforms. This increases knowledge and awareness of Chainlink solutions across the wider community and builds the contributor's personal skills and brand in the community.

Some examples of the content generated from the community:

- Document your experience in using Chainlink as part of your project
- Do a deep dive blog post or video on a Chainlink solution
- Write up technical tutorials showcasing Chainlink being used in various use cases

Becoming a developer expert

Chainlink Developer Experts are smart contract and blockchain developers with deep experience building applications using Chainlink. They are passionate about sharing their technical knowledge with the world. As a developer expert, you will receive recognition in the community, previews of new Chainlink features, exclusive access to Chainlink events, and opportunities to level up your technical and soft skills. You can apply to become a developer expert on the Chainlink Developer Experts page.

Joining the Chainlink Community Advocate program

The Chainlink Community Advocate Program is a program designed to help accelerate the awareness and adoption of Chainlink. Chainlink community advocates are passionate members of the Chainlink community that help to achieve this by running virtual and in-person meetups, connecting with partners and sponsors, creating content, and working directly with the teams that are making Chainlink-powered smart contracts. Many Advocates have gone on to have rewarding careers in the blockchain industry, and some of them work on Chainlink specifically.

To become a community advocate, you can apply via the community advocates web page.

Running a Chainlink Focused Developer Bootcamp

In June 2021, Chainlink virtually hosted the first Chainlink Developer Bootcamp. If you're passionate about educating others about smart contracts and Chainlink, you can contribute by running your own developer Bootcamp. You can also contribute by translating an existing Bootcamp and running it in another language. Before you run your own Bootcamp, reach out to the team so we can make sure you have the support that you need.

Running an in-person meetup or watch party

If you're passionate about helping to grow the awareness and adoption of Chainlink, you can contribute by running an in-person meetup or watch party for a Chainlink event such as SmartCon. Meetups are a great way to meet others also passionate about how hybrid smart contracts can create an economically fair world.

If you're interested in running an in-person meetup or watch party, reach out to the team so we can make sure you have the support that you need.

Participate in a hackathon

Chainlink runs hackathons multiple times per year and often sponsors other hackathons across the blockchain ecosystem. Participating in a hackathon that Chainlink is a part of is a great way to learn how to use Chainlink. It is also a great way to showcase your skills to the Chainlink team and the wider community. Hackathons are a popular place for recruiting talent into the blockchain ecosystem.

To stay up to date on the hackathons that Chainlink is running or sponsoring, keep an eye out on the official Chainlink social media channels, and sign up for our developer newsletter.

Applying for a grant

The Chainlink grant program encourages development of critical developer tooling, add high-quality data, and the launch key services around the Chainlink Network. Grant categories include community, integration, bug bounty, research, and social impact grants. If you have a great idea that fits into one of these categories, you can apply for a grant. If successful, you will receive the funding and support needed to successfully build and implement your idea.

For more information about the grant program, go to the Chainlink Grants web page.

```
# create-a-chainlinked-project.mdx:
```

```
---
section: global
date: Last Modified
title: "Install Frameworks"
whatsnext:
  {
    "Introduction to Data Feeds": "/data-feeds",
    "Introduction to Chainlink VRF": "/vrf",
    "Introduction to Using Any API": "/any-api/introduction",
  }
---
```

```
import { Aside } from "@components"
```

```
!Starter kit logos
```

You can install and use Chainlink in your projects either manually or by using the Chainlink Starter Kits. Once you have the Chainlink library installed, you can more easily access the Chainlink ecosystem.

```
<Aside type="note" title="Important">
```

If you're new to smart contract development and want a step-by-step guide, try out our Getting Started guide.

```
</Aside>
```

Installing into existing projects

Chainlink is supported by Hardhat, Brownie, Truffle, and other frameworks.

If you already have a project, install the @chainlink/contracts NPM package.

NPM

Install using NPM:

```
shell npm
npm install @chainlink/contracts --save
```

Yarn

Install using Yarn:

```
shell yarn
yarn add @chainlink/contracts
```

Create a new project

If you're creating a new project from scratch, these commands will help you set up your project to interact with Chainlink tools and features via the use of our Starter Kits.

Hardhat Starter Kit

For the latest instructions, see the following repositories:

- Hardhat Starter Kit
- Hardhat Starter Kit (TypeScript)

To learn more about Hardhat, read the [Hardhat Documentation](#).

For more details on how to use Chainlink with Hardhat, see the blog post for [How to use Hardhat with Chainlink](#).

Brownie Starter Kit

For the latest instructions, see the [Brownie Starter Kit repository](#).

To learn more about Brownie, read the [Brownie Documentation](#).

For more details on how to use Chainlink with Brownie, see the [Develop a DeFi Project Using Python](#) blog post.

Truffle Starter Kit

For the latest instructions, see the [Truffle Starter Kit repository](#).

To learn more about Truffle, read the [Truffle Suite Documentation](#).

For more details on how to use Chainlink with Truffle, see our blog post about [Using Truffle to interact with Chainlink Smart Contracts](#).

Foundry Starter Kit

For the latest instructions, see the following repositories:

- Foundry Starter Kit
- Foundry Starter Kit (Huff)

To learn more about Foundry, read the [Foundry Documentation](#).

Apeworx Starter Kit (Vyper)

For the latest instructions, see the [Apeworx Starter Kit repository](#).

To learn more about Truffle, read the [Apeworx Documentation](#).

Anchor Starter Kit (Solana)

For the latest instructions, see the [Chainlink Solana Starter Kit repository](#).

To learn more about Anchor, see the [Anchor Documentation](#).

Testing Chainlink contracts

See our blog post on Testing Chainlink Smart Contracts or watch the Chainlink Hackathon Workshop.

Tests samples can be found on Hardhat Starter Kit and Truffle Starter Kit respectively.

```
# developer-communications.mdx:
```

```
---
section: global
date: Last Modified
title: "Developer Communications"
metadata:
  title: "Developer Communications"
  description: "We are committed to communicating these changes with you in
advance. This page will provide information on our current communication
channels and detail active notifications / upgrade plans with timelines."
---
```

```
import DeveloperCommunicationsCallout from
"@features/resources/components/DeveloperCommunicationsCallout.astro"
```

The Chainlink Developer mailing list is the best place to stay up to date on

- Releases
- Package Updates
- New Features
- Breaking Changes
- Events
- Connecting with other developers

```
<DeveloperCommunicationsCallout />
```

```
# fund-your-contract.mdx:
```

```
---
section: global
date: Last Modified
title: "Fund Your Contracts"
---
```

```
import { Aside, ClickToZoom } from "@components"
```

Some smart contracts require funding at their addresses so they can operate without you having to call functions manually and pay for the transactions through MetaMask. This guide explains how to fund Solidity contracts with LINK or ETH.

Retrieve the contract address

1. In Remix, deploy your contract and wait until you see a new contract in the Deployed Contracts section.
1. On the left side panel, use the Copy button located near the contract title to copy the contract address to your clipboard.

```
<ClickToZoom src="/files/25d2c8e-ScreenShot2020-09-08at7.15.50AM.png" />
```

Send funds to your contract

1. Open MetaMask.
1. Select the network that you want to send funds on. For example, select the Sepolia testnet.

1. Click the Send button to initiate a transaction.
1. Paste your contract address in the address field.
1. In the Asset drop down menu, select the type of asset that you need to send to your contract. For example, you can send LINK. If LINK is not listed, follow the guide to Acquire testnet LINK.
1. In the Amount field, enter the amount of LINK that you want to send.
1. Click Next to review the transaction details and the gas cost.
1. If the transaction details are correct, click Confirm and wait for the transaction to process.

<ClickToZoom src="/files/867073d-metamask.png" />

<Aside type="caution" title="Transaction fee didn't update?">

You may need to click Fastest, Fast, Slow, or Advanced Options after entering the Amount to update the gas limit for the token transfer to be successful.

</Aside>

getting-help.mdx:

```
---
section: global
date: Last Modified
title: "Getting Help"
---
```

If you run into issues and the available documentation, videos, and code repositories are not able to assist you, the best way to get help is to follow the support escalation process in this document. Sometimes you might have a question that is too theoretical or hasn't been solved, so you might not always get what you're looking for!

Double check the documentation

Check to see if you missed any code, documentation, blog, or video on the topic or issue you're looking for. There are typically a few different resources on a topic if one doesn't answer exactly what you're looking for. You can also use the documentation search bar to look up things as well.

Do a web search for the specific error or situation you're in

Often someone else has asked the same question that you're asking. If you copy and paste the error into the Google or web search bar, there is a good chance that you will find some helpful material from someone else who has already found the solution to your question.

Open an issue on GitHub or the code repository

This is only applicable if you're working with a certain set of code. For example, if you're having an issue working with the Chainlink Hardhat Starter kit, open an issue on the repo explaining exactly what's going on and someone might have the answer that you need.

When writing issues, remember to:

- Keep titles short
- Be clear and concise about the issue that you are encountering
- Format your issue description. Use three backticks (```) to format your code or log output.
- Always add any and all associated code
- Don't use screenshots. Screenshots are not searchable and generally make it harder to understand your issue.

Ask a question on Stack Overflow or Stack Exchange Ethereum

This is where most people will end up and is one of the most helpful resources out there. Stack Overflow is living documentation, so do your best to make a thoughtful and easy to triage question. This will make it much easier for people to help debug your issue and ensure it doesn't get removed from the site. Remember, we want to make this question searchable so others who run into the same issue can also get their question solved. You could use any forum-based site you like if you prefer another site over Stack Overflow, such as Stack Exchange Ethereum.

It's best to create a minimum reproducible example to help others understand your issue. This way, they can help you get an answer quickly. Remember, it's a community-run platform! Don't get discouraged if your question gets downvoted or removed. This just means you need to format your question a little differently next time!

An example of a poorly formatted question:

Title: Please help

I'm following this guide, and my code is breaking, what's going on?

<https://docs.chain.link/>

Here is my code

```
pragma solidity 0.6.7; contract HelloWorld { string public message;  
constructor(string memory initialMessage) {message = initialMessage; }
```

The same question with better formatting:

Title: Remix Solidity Compile Error - Source File Requires Different Compiler Version

I'm following this guide, and I'm unable to compile my solidity code in Remix.

Here is the code:

```
solidity  
pragma solidity 0.6.7;  
  
contract HelloWorld {  
    string public message;  
  
    constructor(string memory initialMessage) {  
        message = initialMessage;  
    }  
  
    function updateMessage(string memory newMessage) public {  
        message = newMessage;  
    }  
}
```

And the error I'm getting is as follows:

```
ParserError: Source file requires different compiler version (current compiler  
is 0.8.7+commit.e28d00a7.Emscripten.clang) - note that nightly builds are  
considered to be strictly less than the released version
```

1

Ask the community

You can always ask the question in the Discord and see if there is a community member who might be able to help you out. One of the best ways to ask the community is to drop a link to your Stack Overflow question, issue, or the forum where you're asking a Chainlink question. Remember, these are community members, and they are helping because they are wonderful and kind individuals!

For important updates regarding the use of Chainlink Price Feeds, users should join the official Chainlink Discord and subscribe to the data-feeds-user-notifications channel: <https://discord.gg/Dqy5N9UbsR>

```
# glossary.mdx:
```

```
---
date: Last Modified
title: "Glossary"
section: global
---
```

```
import { Aside } from "@components"
```

Adapter

```
<Aside type="danger" title="">
  The adapters or JSON adapters for v1 Jobs are removed for Chainlink nodes
  running version 1.0.0 and later. Use v2 job
  tasks instead.
</Aside>
```

An adapter or task is a piece of software responsible for executing a specific piece of functionality. A Chainlink node comes with a number of Adapters built-in, commonly known as Core Adapters, but can also be extended via Bridges to connect with user-defined External Adapters.

Answer

The result produced from an oracle service, after all safety checks and aggregations have been performed.

Bridge

Bridge is the connection between a Chainlink node and an External Adapter. The External Adapter runs as a separate service, and a Bridge facilitates communication between the node and one of these adapters.

If you would like to add a new External Adapter to your node, you create a new Bridge either in the GUI or the CLI. Within the Chainlink node, a bridge must have a unique name, but can share the same URL with other bridges. You can also set a different number of default confirmations for each bridge, and an additional payment amount. Once the bridge is added to the node, its name can then be used as a task type in Jobs.

Consumer (Contract)

Recipient of an Answer provided by an Oracle. The Consumer is commonly a

contract, and is also commonly the same entity that requested the Answer, but does not have to be. We have a helper function, `addExternalRequest`, that gives consuming contracts the ability to safely check answers it receives without requesting them itself.

Encumbrance parameters

Encumbrance parameters are the part of a service agreement that can be enforced onchain. Information on encumbrance parameters can be found [on our Wiki](https://github.com/smartcontractkit/chainlink/wiki/Service-Agreements-and-the-Coordinator-Contract).

External adapter

External adapters are what make Chainlink easily extensible, providing simple integration of custom computations and specialized APIs.

A Chainlink node communicates with external adapters by sending a POST request with a JSON data payload. More information can be found on the external adapter page.

Function selector

A function selector specifies the function to be called in Ethereum. It is the first four bytes of the call data for a function call in an Ethereum transaction. Solidity contracts have a built-in helper method to access the function selector by using `this.myFunction.selector`, where `myFunction` is a non-overloaded function in the contract.

Initiator

<Aside type="danger" title="">
The initiators for v1 Jobs are removed for Chainlink nodes running version 1.0.0 and later. Use the v2 job types instead.
</Aside>

Triggers the execution of a Job Spec.

Job

Short-hand for a Job Spec.

Job run

The Job Run is the artifact documenting the outcome of executing a Job. The Job Run is made up of a Task and a Run Result representing the ultimate outcome of the Job Run.

JobID

The ID associated to a given Job Spec. This will be unique per-node, even with the same contents within the spec itself.

Job spec

The Job Specification is the specification of a piece of work to be completed by an Oracle Node. The Job Spec is made up of two main parts:

- The Task Type or the External Initiator: Defines the ways a Job can be triggered to execute.
- The Task list: The tasks that specify all of the computation steps to perform when executing a Job Spec. The Task list is sometimes referred to as the Job Pipeline because all of the Tasks' operations are performed in order, with the

result being fed into the next task.

Oracle

Entity which connects computations on blockchains with offchain resources. Typically made up of two components: the Oracle Node (offchain) and the Oracle Contract (onchain).

Oracle contract

The onchain component of an Oracle. The Oracle Contract is the interface through which Consuming Contracts pass and receive data with offchain resources.

Oracle node

The offchain component of an Oracle.

Phase

For data feeds, a phase indicates the underlying aggregator implementation has been updated. Phases are relevant only for the EACAggregatorProxys. You can think of a roundId on the proxies as a large number containing data for two numbers (phaseId + roundId). The roundId is pulled from the aggregator's implementation and combined by bit shifting with the latest phaseId of the proxy.

Requester

A Smart Contract or Externally Owned Account which requests data from an Oracle. The Requester does not have to be the same entity as the Consumer but commonly is the same.

Run result

A Run Result is the result of executing a Job Spec.

Run status

Each Job Run has a status field indicating its current progress. The Run Status can be in one of the following states:

- Unstarted
- In Progress
- Pending Confrimations
- Pending Bridge
- Pending Sleep
- Errored
- Completed

SAID

The ID associated with a given Service Agreement.

Service agreement

The Service agreement consists of a Job Spec and a set of encumbrance parameters that is shared among a creator and multiple Chainlink nodes. Information on service agreements can be found on our Wiki.

Spec

Another short-hand for a Job Spec.

Task

A v2 job task.

Task spec

The Task Spec is the definition for an individual task to be performed within the job specification by a specific adapter. The Task Spec always includes a type field which specifies which adapter will execute it. Optionally, a Task Spec can specify additional params which will be passed on to its adapter, and confirmations which specify how many confirmations a Task Run needs before executing.

Task run

The result of the individual Task Spec's execution. A Task Run includes the Task Spec that it used for input and the Run Result which was the output of the execution.

```
# hackathon-resources.mdx:
```

```
---
section: global
date: Last Modified
title: "Hackathon Resources"
---
```

```
import { Aside } from "@components"
```

```
<Aside type="note" title="Note on Resources">
```

For a comprehensive list of resources, refer to the Learning Resources page.

```
</Aside>
```

This page lists useful resource to help you get started with Hackathon projects. If you want to check out code from past hackathons to get some inspiration, check out the Blog to find past hackathons and winners.

Starter kits

You can use the starter kits to help test, deploy, interact with, and maintain your smart contracts. Starter kits are available for several different languages and frameworks. You can see the full list of available starter kits in the Starter Kits GitHub repository.

Support communications

Always refer to the getting help page for the latest information about how to get support.

- Getting Help
- Stack Overflow
- Stack Exchange Ethereum
- Hackathon Discord
- Developer Discord

Tutorials

If you are new to Smart Contracts, read the Getting Started Guide.

- Learning Resources
- Video Tutorials
- What is Ethereum?

- Developer Blog featuring several tutorials
- Testing Chainlink Smart Contracts
- NFTs and Chainlink
- Build an external adapter

Inspiration

- 77 Use Cases by Chainlink
- Blog posts featuring past hackathons and winners

Partners and BUILD projects

Partners

- CTOR Labs
- Quicknode
- Covalent
- 0xcord

BUILD projects

- Cryptum
- Source Network
- Thirdfi

Join the community

The Chainlink community is some of the most inviting groups of engineers always looking to help you grow to the next stage.

- Twitter
- Reddit
- Telegram
- Blog

hackathon-rules-waiver-and-release.mdx:

```
---
section: global
date: Last Modified
title: "Hackathon Rules, Waiver & Release, and Code of Conduct"
---
```

Code of Conduct

This event is a community hackathon intended for collaboration and learning in the Chainlink and broader blockchain communities. We value the participation of each member of the community and want all participants to have an enjoyable experience. Accordingly, all participants are expected to show respect and courtesy to other participants throughout the hackathon. To make clear what is expected, all participants of this hackathon are required to conform to the following Code of Conduct. Organizers will enforce this code throughout the event.

The Code

The spirit of The Code is to prohibit activities including but not limited to:

- Comments that others find offensive
- Cheating or taking unfair advantage of other participants' work or efforts
- Any activity related to harassing, demeaning, mocking, or intimidating others, especially this behavior as it relates to characteristics such as:
 - Gender

- Sexual orientation
- Physical or mental ability
- Age
- Socioeconomic status
- Ethnicity
- Physical appearance
- Race
- Religion
- Country of origin
- Examples of other prohibited behaviors include, but are not limited to:
 - Stalking
 - Unwanted sexual attention
 - Use of sexualized content

Participants asked to stop any behavior deemed as harassment are expected to comply immediately. If a participant fails to comply they will be asked to leave the event. Sponsors, judges, mentors, volunteers, organizers, Chainlink team, and anyone else at the event are also subject to the Code.

If a participant engages in behavior that violates the Code, the hackathon organizers will take any action they deem appropriate, including warning the offender or expelling them from the event.

If you feel uncomfortable or think there may be a potential violation of the code of conduct, please report it immediately to one of the event organizers or by emailing us at legal@chain.link. All reporters have the right to remain anonymous.

Event Rules & Conditions

1. The following rules & conditions (the "Rules") apply to this Chainlink hackathon (the "Event"). By clicking the "I Accept" button, you acknowledge that you have read these Rules, understand them, and agree to be bound as follows:

1. You assume full responsibility for any damage or injury caused by you in your participation in the Event (whether to persons or property, and whether to yourself or others) and release SmartContract Chainlink Limited SEZC (and its affiliates) ("Chainlink"), the Event, the organizers of the Event (the "Organizers"), all sponsors of the Event ("Sponsors"), the Event volunteers and the Event staff, (collectively, the "Releasees") from any liability therefore. YOU ARE AWARE THAT YOUR PARTICIPATION IN THE EVENT IS SOLELY AT YOUR OWN RISK, AND THAT THE RELEASE HEREIN IS INTENDED TO REFLECT THAT UNDERSTANDING

1. You will own any developments that you create during the Event, and all right, title and interest in those developments, including the intellectual property rights therein, shall belong to you. However, you acknowledge that during the course of the Event, you may obtain access to products, developments, information and other materials belonging to Chainlink, other participants of the Event, the Sponsors and/or other third parties ("Third Party Materials"), and that nothing in this Agreement is deemed to transfer any ownership, right, title or interest in such Third Party Materials to you. Your only rights to the Third Party Materials shall be those expressly granted to you by the owner(s) of the Third Party Materials. Specifically, any APIs or other software provided to you by the Sponsors are subject to the subscription terms and software licenses associated with such APIs or other software.

1. By entering this Event, you represent and warrant that your participation complies with these Rules and that you have sufficient rights to (1) authorize the publication and dissemination of any submission materials and presentations ("Submitted Materials"); (2) allow the Organizers and Sponsors to use and to authorize others to use, publish and disseminate your Submitted Materials. Further, you are entirely responsible for your Submitted Materials, in whole or in part, if: (a) determined to be defamatory, offensive or otherwise inappropriate; (b) determined to violate any laws, rules or regulations; (c) determined to be infringing, or constitute a misappropriation of any

intellectual property rights or confidential or proprietary information of any third party; or (d) determined to violate these Rules. Your Submitted Materials must be true and accurate and in compliance with these Rules in all regards. At any time, the Organizers at their sole discretion, reserve the right to remove your Submitted Materials from the Event, in whole or in part, for any violation of these Official Rules.

1. You acknowledge that the Event is intended to be a place where ideas are shared freely, and therefore acknowledge that any information that you share with other participants of the Event, the Sponsors and/or other third parties during the Event is solely at your discretion and risk. If you wish to protect your information, it is solely your responsibility to implement confidentiality and security measures with respect to the persons to whom you are disclosing your information. None of the Releasees shall have any responsibility under this Agreement or by virtue of their participation in the Event with respect to your information.

1. You are not be a citizen or resident of any jurisdiction subject to sanctions as enforced by the Office of Foreign Assets Control, including without limitation Burma, Crimea and Sevastopol, Cote d'Ivoire, Cuba, Democratic Republic of Congo, Iran, Iraq, Libya, North Korea, Sudan, Syria, Zimbabwe, and you must not be named by OFAC as a Specially Designated National or Blocked Person which can be found at: www.treasury.gov

1. You acknowledge that the Organizers and Chainlink have the right to reject participants in the Event at their sole discretion.

1. The Organizers have the unrestricted right to use your likeness, image, voice, opinions, and appearance, and also any images of your projects, developments, materials and belongings made at or brought to the Event, captured through video, photographs or other media during the Event for the express purpose of creating promotional material (the "Images"), for the purposes of use in websites, promotional materials, publications and other media of any of the Organizers, whether in print or electronically (the "Materials"). The foregoing right includes permission to copyright, use, re-use, publish, and republish Images in which you may be included, intact or in part, composite or distorted in character or form, without restriction as to changes or transformations, in conjunction with your own or a fictitious name, reproduction in color or otherwise, made through any and all media now or hereafter known;

1. The Organizers shall solely own the Materials in which you or your Images, in whole or in part, may appear, including copyright interests, and you have no ownership rights therein;

1. You give all clearances, copyright and otherwise, for use of your Images, and waive any moral rights that you may have in the Materials in which you or your Images may appear. The rights granted to the Organizers herein are perpetual and worldwide. For greater certainty, you agree that your images may continue to be used after the completion of the Event;

1. You relinquish any right that you may have to examine or approve the Materials in which you or your Images may appear or the use to which they may be applied; and

1. You hereby release, discharge and agree to save harmless each and all of the Organizers from any liability by virtue of any blurring, distortion, alteration, optical illusion, or use in composite form of the Images whether intentional or otherwise, that may occur or be produced in the recording of the Images or in any subsequent processing thereof, as well as any publication thereof, including without limitation any claims for libel or invasion of privacy.

1. You agree that the Organizers may share your registration details, LinkedIn/Github profiles, details of your Hackathon submission, and other information obtained from you in the course of, or relating to, the Event with the Sponsors, and acknowledge that such Sponsors may contact you during and after the Event. By agreeing to this document and/or participating in the Event, you are providing your express consent to communications by the Organizers and Sponsors (including email communications, both marketing and informational) respecting the products and services of the Organizers and Sponsors, and future events.

1. For valuable consideration, including permission to take part in the Event, you hereby covenant not to sue, and release, waive, and discharge the Releasees,

their owners, officers, agents, affiliates, employees, volunteers, and/or any other person or entity in any way associated with the Event, from liability for any injury to your person or property or death arising out of or related to your participation in the Event, whether caused by an act of negligence of the Releasees or otherwise; and hereby assume full responsibility for any risk of bodily injury, death or property damage arising out of or related to your participation in the Event, whether occurring to you or to any other person or entity for whom you are responsible or with whom you are associated, and whether caused by an act of negligence of the Releasees or otherwise. The foregoing release includes, but is not limited to, any occurrences of personal injury, illness (food-borne or otherwise), and loss of belongings, whether by theft or otherwise. You further agree that this instrument (the terms of which collectively are referred to as the Rules) is intended to be as broad and inclusive as is permitted by the laws of the State of California and that if any portion thereof is held invalid, that portion shall be invalid only to the extent required by law, and the balance shall, notwithstanding, continue in full force and effect.

1. You agree to indemnify and hold the Organizers and Sponsors (and judges, mentors, volunteers, organizers, Chainlink team administering the Event) and each of their employees, representatives, agents, attorneys, affiliates, directors, employees, officers, managers, and shareholders (the "Indemnified Parties") harmless from any damage, loss, cost, or expense (including without limitation, attorneys' fees and costs) incurred in connection with any third-party claim, demand, or action ("Claim") brought or asserted against any of the Indemnified Parties, alleging facts or circumstances that would constitute a breach of any provision of these Rules by you; arising from, related to, or connected with your entry, Submitted Materials, presentations and participation in any way in any aspect of Event, including receipt of any prize. If you are obligated to provide indemnification pursuant to this provision, the Indemnified Parties may, in their sole discretion, control the disposition of any claim at your sole cost and expense. Without limitation of the foregoing, you may not settle, compromise, or in any other manner dispose of any claim without the Organizers' express written consent.

1. If selected as a winner of a Chainlink award, your acceptance of the Chainlink award means you agree to the following:

1. You will not disparage Chainlink or its products, services, agents, representatives, directors, officers, shareholders, attorneys, employees, vendors, business partners, affiliates, successors or assigns, or any person acting by, through, under or in concert with any of them, with any written or oral statement. Nothing in this paragraph shall prohibit the winner from providing truthful information in response to a valid subpoena or other legal process; however, the winning participant agrees to provide Company sufficient notice of such to allow Company the opportunity to oppose such subpoena or legal process prior to providing any information (unless expressly prohibited by applicable law).

1. You agree that awards are subject to availability and Chainlink and the Sponsors reserve the right to substitute or withdraw any prize without giving notice at their sole discretion.

1. Any media, such as a blog post, that is created about winning the Event, will first be shared with the Chainlink team beforehand so that we can coordinate and help you spread the message.

1. You agree to an interview with a writer affiliated with the Chainlink team, which will result in featuring your project on the Chainlink website and sharing your project with the greater community.

link-token-contracts.mdx:

section: global

date: Last Modified

title: "LINK Token Contracts"

metadata:

title: "LINK Token Contracts"

```
description: "Addresses for the LINK token on supported networks."
linkToWallet: true
image: "/files/72d4bd9-link.png"
```

```
import { Aside, Address } from "@components"
import ResourcesCallout from
"@features/resources/callouts/ResourcesCallout.astro"
import CcipCommon from "@features/ccip/CcipCommon.astro"
```

```
<Aside type="note" title="Talk to an expert">
  <a href="https://chain.link/contact?refid=Contracts">Contact us</a> to talk to
an expert about the networks that
  support the LINK token.
</Aside>
```

LINK tokens are used to pay node operators for retrieving data for smart contracts and also for deposits placed by node operators as required by contract creators. The smallest denomination of LINK is called a Juel, and 1,000,000,000,000,000,000 (1e18) Juels are equal to 1 LINK. This is similar to Wei, which is the smallest denomination of ETH.

The LINK token is an ERC677 token that inherits functionality from the ERC20 token standard and allows token transfers to contain a data payload. Read more about the ERC677 transferAndCall token standard.

To use Chainlink services on a given blockchain, it is necessary to use LINK tokens. You can transfer tokens across blockchains by using Chainlink CCIP or applications such as Transporter and XSwap.

```
<CcipCommon callout="thirdPartyApps" />
<ResourcesCallout callout="bridgeRisks" />
```

Ethereum

Ethereum mainnet

ETH is used to pay for transactions on Ethereum Mainnet.

Parameter	Value
:-----	
:-----	

Chain ID	1
Address	<Address
contractUrl="https://etherscan.io/token/0x514910771AF9Ca656af840dff83E8264EcF986CA" urlId="10x514910771AF9Ca656af840dff83E8264EcF986CA" urlClass="erc-token-address"/>	
Name	Chainlink Token
Symbol	LINK
Decimals	18
Network status	ethstats.dev

Sepolia testnet

Testnet ETH is used to pay for transactions on Sepolia.

Testnet ETH and LINK are available at faucets.chain.link/sepolia.

Parameter	Value
:-----	
:-----	

Chain ID	11155111
Address	<Address contractUrl="https://sepolia.etherscan.io/token/0x779877A7B0D9E8603169DdbD7836e478b4624789" urlId="111551110x779877A7B0D9E8603169DdbD7836e478b4624789" urlClass="erc-token-address"/>
Name	Chainlink Token
Symbol	LINK
Decimals	18
Network status	etherscan.freshstatus.io

BNB Chain

BNB Chain mainnet

BNB is used to pay for transactions on the BNB Chain Mainnet.

<Aside type="caution" title="ERC-677 LINK on BNB Chain">

The LINK provided by the BNB Chain Bridge is not ERC-677 compatible, so you cannot use it with Chainlink services or oracle nodes. Use the Chainlink PegSwap service to convert bridged LINK to the official ERC-677 LINK token on BNB Chain.

</Aside>

Parameter	Value
:-----	
:-----	

Chain ID	56
Address	<Address contractUrl="https://bscscan.com/token/0x404460C6A5EdE2D891e8297795264fDe62ADBB75" urlId="560x404460C6A5EdE2D891e8297795264fDe62ADBB75" urlClass="erc-token-address"/>
Name	Chainlink Token
Symbol	LINK
Decimals	18
Network status	bscscan.freshstatus.io

BNB Chain testnet

Testnet BNB is used to pay for transactions on the BNB Chain testnet. Testnet BNB is available at testnet.bnbchain.org/faucet-smart.

Testnet LINK is available at faucets.chain.link/bnb-chain-testnet.

Parameter	Value
:-----	
:-----	

Chain ID	97
Address	<Address contractUrl="https://testnet.bscscan.com/address/0x84b9B910527Ad5C03A9Ca831909E21e236EA7b06" urlId="970x84b9B910527Ad5C03A9Ca831909E21e236EA7b06" urlClass="erc-token-address"/>
Name	Chainlink Token
Symbol	LINK
Decimals	18
Network status	bscscan.freshstatus.io

Polygon

Polygon mainnet

POL is used to pay for transactions on Polygon. You can use the Polygon Bridge to transfer tokens to Polygon Mainnet and then use Polygon Gas Swap to swap supported tokens to POL.

<Aside type="caution" title="ERC-677 LINK on Polygon">

The LINK provided by the Polygon Bridge is not ERC-677 compatible, so you cannot use it with Chainlink services or oracle nodes. Use the Chainlink PegSwap service to convert bridged LINK to the official ERC-677 LINK token on Polygon.

Watch the Moving Chainlink Cross-Chains video to learn more.

</Aside>

Parameter	Value
:-----	
:-----	

Chain ID	137
Address	<Address contractUrl="https://polygonscan.com/address/0xb0897686c545045aFc77CF20eC7A532E3120E0F1" urlId="1370xb0897686c545045aFc77CF20eC7A532E3120E0F1" urlClass="erc-token-address"/>
Name	Chainlink Token
Symbol	LINK
Decimals	18
Network status	polygonscan.freshstatus.io

Amoy testnet

Testnet POL is used to pay for transactions on Polygon Amoy.

Testnet POL and LINK are available at faucets.chain.link/polygon-amoy.

Parameter	Value
:	-----
:	-----

Chain ID	80002
Address	<Address contractUrl="https://amoy.polygonscan.com/address/0x0Fd9e8d3aF1aeee056EB9e802c3A762a667b1904" urlId="800020x0Fd9e8d3aF1aeee056EB9e802c3A762a667b1904" urlClass="erc-token-address"/>
Name	Chainlink Token
Symbol	LINK
Decimals	18
Network status	polygonscan.freshstatus.io

Gnosis Chain (xDai)

Gnosis Chain mainnet

xDAI is used to pay for transactions on Gnosis Chain Mainnet. Use the xDai Bridge to send DAI from Ethereum Mainnet to Gnosis Chain and convert it to xDAI. Use OmniBridge to send LINK from Ethereum Mainnet to Gnosis Chain.

Parameter	Value
:	-----
:	-----

Chain ID	100
Address	<Address contractUrl="https://gnosisscan.io/address/0xE2e73A1c69ecF83F464EFCE6A5be353a37cA09b2" urlId="1000xE2e73A1c69ecF83F464EFCE6A5be353a37cA09b2" urlClass="erc-token-address"/>
Name	Chainlink Token on Gnosis Chain (xDai)
Symbol	LINK
Decimals	18
Network status	gnosisscan.freshstatus.io

Gnosis Chiado testnet

xDAI is used to pay for transactions on Gnosis Chiado testnet. Use the Chiado faucet to get testnet xDAI.

Testnet LINK is available at faucets.chain.link/gnosis-chiado-testnet.

Parameter	Value
:	-----
:	-----

```

-----
| Chain ID          | 10200
|
| Address           | <Address
contractUrl="https://gnosis-chiado.blockscout.com/address/0xDCA67FD8324990792C0bfaE95903B8A64097754F" urlId="102000xDCA67FD8324990792C0bfaE95903B8A64097754F"
urlClass="erc-token-address"/> |
| Name              | Chainlink Token on Gnosis Chiado testnet
|
| Symbol            | LINK
|
| Decimals          | 18
|
| Network status    | gnosisscan.freshstatus.io
|

```

Avalanche

Avalanche mainnet

AVAX is used to pay for transactions on Avalanche Mainnet. Use the Avalanche Bridge to transfer LINK from Ethereum Mainnet to Avalanche.

```

| Parameter          | Value
|
| :-----
| :-----
-----
| Chain ID          | 43114
|
| Address           | <Address
contractUrl="https://snowtrace.io/address/0x5947BB275c521040051D82396192181b413227A3" urlId="431140x5947BB275c521040051D82396192181b413227A3" urlClass="erc-
token-address"/> |
| Name              | Chainlink Token on Avalanche
|
| Symbol            | LINK
|
| Decimals          | 18
|
| Network status    | status.avax.network
|

```

Fuji testnet

Testnet AVAX is used to pay for transactions on Avalanche Fuji.

Testnet AVAX and LINK are available at faucets.chain.link/fuji. Testnet AVAX is also available at core.app/tools/testnet-faucet.

```

| Parameter          | Value
|
| :-----
| :-----
-----
| Chain ID          | 43113
|
| Address           | <Address
contractUrl="https://testnet.snowtrace.io/address/0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" urlId="431130x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846"
urlClass="erc-token-address"/> |

```

Name	Chainlink Token on Avalanche
Symbol	LINK
Decimals	18
Network status	status.avax.network

Fantom

Fantom mainnet

FTM is used to pay for transactions on Fantom Mainnet.

<Aside type="caution" title="ERC-677 LINK on Fantom">

You must use ERC-677 LINK on Fantom. ERC-20 LINK will not work with Chainlink services.

</Aside>

Parameter	Value
:-----	
:	-----

Chain ID	250
Address	<Address contractUrl="https://ftmscan.com/address/0x6F43FF82CCA38001B6699a8AC47A2d0E66939407" urlId="2500x6F43FF82CCA38001B6699a8AC47A2d0E66939407" urlClass="erc-token-address"/>
Name	Chainlink Token on Fantom
Symbol	LINK
Decimals	18
Network status	ftmscan.freshstatus.io

Fantom testnet

Testnet FTM is used to pay for transactions on Fantom testnet. Testnet FTM is available at faucet.fantom.network.

Testnet LINK is available at faucets.chain.link/fantom-testnet.

Parameter	Value
:-----	
:	-----

Chain ID	4002
Address	<Address contractUrl="https://testnet.ftmscan.com/address/0xfaFedb041c0DD4fA2Dc0d87a6B0979Ee6FA7af5F" urlId="40020xfaFedb041c0DD4fA2Dc0d87a6B0979Ee6FA7af5F" urlClass="erc-token-address"/>
Name	Chainlink Token on Fantom

Symbol	LINK
Decimals	18
Network status	ftmscan.freshstatus.io

Arbitrum

Arbitrum mainnet

ETH is used to pay for transactions on the Arbitrum Mainnet.

You can use the Arbitrum Bridge to transfer ETH and LINK from Ethereum Mainnet to Arbitrum Mainnet.

Parameter	Value
:-----	-----
:	-----
-----	-----
Chain ID	42161
Address	<Address contractUrl="https://explorer.arbitrum.io/address/0xf97f4df75117a78c1A5a0DBb814Af92458539FB4" urlId="421610xf97f4df75117a78c1A5a0DBb814Af92458539FB4" urlClass="erc-token-address"/>
Name	Chainlink Token on Arbitrum Mainnet
Symbol	LINK
Decimals	18
Network status	arbiscan.freshstatus.io

Arbitrum Sepolia testnet

Testnet ETH is used to pay for transactions on Arbitrum Sepolia.

Testnet ETH and LINK are available at faucets.chain.link/arbitrum-sepolia.

Parameter	Value
:-----	-----
:	-----
-----	-----
Chain ID	421614
Address	<Address contractUrl="https://sepolia.arbiscan.io/address/0xb1D4538B4571d411F07960EF2838Ce337FE1E80E" urlId="4216140xb1D4538B4571d411F07960EF2838Ce337FE1E80E" urlClass="erc-token-address"/>
Name	Chainlink Token on Arbitrum Sepolia
Symbol	LINK
Decimals	18
Network status	arbiscan.freshstatus.io

Optimism

Optimism mainnet

ETH is used to pay for transactions on Optimism. Use the Optimism Bridge to transfer ETH and LINK from Ethereum Mainnet to Optimism Mainnet.

Parameter	Value
:	-----
:	-----

Chain ID	10
Address	<Address contractUrl="https://optimistic.etherscan.io/address/0x350a791Bfc2C21F9Ed5d10980Dad2e2638ffa7f6" urlId="100x350a791Bfc2C21F9Ed5d10980Dad2e2638ffa7f6" urlClass="erc-token-address"/>
Name	Chainlink Token on Optimism Mainnet
Symbol	LINK
Decimals	18
Network status	status.optimism.io

Optimism Sepolia testnet

Testnet ETH is used to pay for transactions on Optimism Sepolia. Use the Optimism Bridge to transfer testnet ETH from Ethereum Sepolia to Optimism Sepolia. Testnet ETH is available at faucets.chain.link/sepolia.

Testnet LINK is available at faucets.chain.link/optimism-sepolia. Testnet bridges might not transfer the correct type of LINK to Optimism Sepolia, so it is recommended to use only the LINK acquired from faucets.chain.link/optimism-sepolia when developing applications on testnet.

Parameter	Value
:	-----
:	-----

Chain ID	11155420
Address	<Address contractUrl="https://sepolia-optimism.etherscan.io/token/0xE4aB69C077896252FAFBD49EFD26B5D171A32410" urlId="111554200xE4aB69C077896252FAFBD49EFD26B5D171A32410" urlClass="erc-token-address"/>
Name	Chainlink Token on Optimism Sepolia
Symbol	LINK
Decimals	18
Network status	status.optimism.io

Moonriver

Moonriver mainnet

MOVR is used to pay transaction fees on Moonriver Mainnet.

Parameter	Value
:-----	
:-----	

Chain ID	1285
Address	<Address contractUrl="https://moonriver.moonscan.io/address/0x8b12Ac23BFe11cAb03a634C1F117D64a7f2cFD3e" urlId="12850x8b12Ac23BFe11cAb03a634C1F117D64a7f2cFD3e" urlClass="erc-token-address"/>
Name	Chainlink Token on Moonriver Mainnet
Symbol	LINK
Decimals	18
Network status	moonscan.freshstatus.io

Moonbeam

Moonbeam mainnet

GLMR is used to pay transaction fees on Moonbeam Mainnet.

Parameter	Value
:-----	
:-----	

Chain ID	1284
Address	<Address contractUrl="https://moonscan.io/address/0x012414A392F9FA442a3109f1320c439C45518aC3" urlId="12840x012414A392F9FA442a3109f1320c439C45518aC3" urlClass="erc-token-address"/>
Name	Chainlink Token on Moonbeam Mainnet
Symbol	LINK
Decimals	18
Network status	moonscan.freshstatus.io

Metis

Metis Andromeda mainnet

METIS is used to pay for transactions on Metis Mainnet. You can use the Metis Bridge to transfer METIS from Ethereum Mainnet to Metis Mainnet.

<Aside type="caution" title="ERC-677 LINK on Metis">

The LINK provided by the Metis Bridge is not ERC-677 compatible, so you cannot use it with

Chainlink services or oracle nodes.

</Aside>

Parameter	Value
-----------	-------

```

| :-----
| :-----
|-----
|----- |
| Chain ID      | 1088
| Address       | <Address
contractUrl="https://explorer.metis.io/address/0xd2FE54D1E5F568eB710ba9d898Bf4bD
02C7c0353" urlId="10880xd2FE54D1E5F568eB710ba9d898Bf4bD02C7c0353" urlClass="erc-
token-address"/> |
| Name          | Chainlink Token on Metis Mainnet
| Symbol        | LINK
| Decimals      | 18
| Network status | explorer.metis.io

```

Metis Sepolia testnet

Testnet METIS is used to pay for transactions on Metis Sepolia.

Testnet METIS and LINK are available at faucets.chain.link/metis-sepolia.

Parameter	Value
Chain ID	59902
Address	<Address contractUrl="https://sepolia-explorer.metisdevops.link/address/0x9870D6a0e05F867EAAe696e106741843F7fD116D" urlId="599020x9870D6a0e05F867EAAe696e106741843F7fD116D" urlClass="erc-token-address"/>
Name	Chainlink Token on Metis Sepolia
Symbol	LINK
Decimals	18
Network status	sepolia-explorer.metisdevops.link

BASE

BASE mainnet

ETH is used to pay for transactions on BASE. You can use the BASE Bridge to transfer ETH from Ethereum Mainnet to BASE Mainnet. To transfer LINK from Ethereum to Base, use Transporter or XSwap Bridge.

Parameter	Value
:	
:	
Chain ID	8453
Address	<Address

```
contractUrl="https://basescan.org/address/0x88Fb150BDc53A65fe94Dea0c9BA0a6dAf8C6e196" urlId="84530x88Fb150BDc53A65fe94Dea0c9BA0a6dAf8C6e196" urlClass="erc-token-address"/> |
| Name                | Chainlink Token on BASE Mainnet
|
| Symbol              | LINK
|
| Decimals            | 18
|
| Network status      | basescan.org
|
```

BASE Sepolia testnet

Testnet ETH is used to pay for transactions on BASE Sepolia.

Testnet ETH and LINK are available at faucets.chain.link/base-sepolia. Testnet ETH is also available from one of the BASE Network Faucets.

```
| Parameter          | Value
|
| :-----
| :-----
|-----
|----- |
| Chain ID           | 84532
|
| Address             | <Address
contractUrl="https://sepolia.basescan.org/address/0xE4aB69C077896252FAFBD49EFD26B5D171A32410" urlId="845320xE4aB69C077896252FAFBD49EFD26B5D171A32410"
urlClass="erc-token-address"/> |
| Name                | Chainlink Token on BASE Sepolia testnet
|
| Symbol              | LINK
|
| Decimals            | 18
|
| Network status      | sepolia.basescan.org
|
```

Blast

Blast mainnet

ETH is used to pay for transactions on Blast. You can use the Blast Bridge to transfer ETH from Ethereum Mainnet to Blast Mainnet.

```
| Parameter          | Value
|
| :-----
| :-----
|-----
|----- |
| Chain ID           | 81457
|
| Address             | <Address
contractUrl="https://blastscan.io/address/0x93202ec683288a9ea75bb829c6bacfb2bfea9013" urlId="814570x93202ec683288a9EA75BB829c6baCFb2BfeA9013" urlClass="erc-token-address"/> |
| Name                | Chainlink Token on Blast Mainnet
|
| Symbol              | LINK
|
| Decimals            | 18
|
```

```
|
| Network status | blastscan.io
|
```

Blast Sepolia testnet

Testnet ETH is used to pay for transactions on Blast Sepolia. Testnet ETH is available from one of the Blast Network Faucets.

```
| Parameter      | Value
| :-----
| :-----
|-----|
| Chain ID       | 168587773
|
| Address        | <Address
contractUrl="https://sepolia.blastscan.io/address/0x02c359ebf98fc8bf793f970f9b83
02bb373bdf32" urlId="1685877730x02c359ebf98fc8BF793F970F9B8302bb373BdF32"
urlClass="erc-token-address"/> |
| Name           | Chainlink Token on Blast Sepolia testnet
|
| Symbol         | LINK
|
| Decimals       | 18
|
| Network status | sepolia.blastscan.io
|
```

Celo

Celo mainnet

CELO is used to pay for transactions on the Celo network.

```
| Parameter      | Value
| :-----
| :-----
|-----|
| Chain ID       | 42220
|
| Address        | <Address
contractUrl="https://explorer.celo.org/mainnet/address/0xd07294e6E917e07dfDcee88
2dd1e2565085C2ae0" urlId="422200xd07294e6E917e07dfDcee882dd1e2565085C2ae0"
urlClass="erc-token-address"/> |
| Name           | Chainlink Token on Celo Mainnet
|
| Symbol         | LINK
|
| Decimals       | 18
|
| Network status | explorer.celo.org
|
```

Celo Alfajores testnet

Testnet CELO is used to pay for transactions on Celo Alfajores. Testnet CELO is available from the Alfajores Token Faucet.

Testnet LINK is available at faucets.chain.link/celo-alfajores-testnet.

Parameter	Value
:	-----
:	-----

Chain ID	44787
Address	<Address contractUrl="https://explorer.celo.org/alfajores/address/0x32E08557B14FaD8908025619797221281D439071" urlId="447870x32E08557B14FaD8908025619797221281D439071" urlClass="erc-token-address"/>
Name	Chainlink Token on Celo Alfajores testnet
Symbol	LINK
Decimals	18
Network status	explorer.celo.org/alfajores

Scroll

Scroll mainnet

ETH is used to pay for transactions on Scroll Mainnet. Use the Scroll Bridge to transfer ETH from Ethereum to Scroll.

Parameter	Value
:	-----
:	-----

Chain ID	534352
Address	<Address contractUrl="https://scrollscan.com/address/0x548C6944cba02B9D1C0570102c89de64D258d3Ac" urlId="5343520x548C6944cba02B9D1C0570102c89de64D258d3Ac" urlClass="erc-token-address"/>
Name	Chainlink Token on Scroll Mainnet
Symbol	LINK
Decimals	18
Network status	status.scroll.io

Scroll Sepolia testnet

Testnet ETH is used to pay for transactions on Scroll testnet.

Testnet ETH and LINK are available at faucets.chain.link/scroll-sepolia-testnet. Testnet ETH is also available from the Scroll Sepolia Faucets.

Parameter	Value
:	-----
:	-----

Chain ID	534351

```
| Address | <Address
contractUrl="https://sepolia-blockscout.scroll.io/address/0x231d45b53C905c3d6201
318156BDC725c9c3B9B1" urlId="5343510x231d45b53C905c3d6201318156BDC725c9c3B9B1"
urlClass="erc-token-address"/> |
| Name | Chainlink Token on Scroll Sepolia testnet
|
| Symbol | LINK
|
| Decimals | 18
|
| Network status | status.scroll.io
```

Linea

Linea mainnet

ETH is used to pay for transactions on Linea Mainnet. Use the Linea Bridge to transfer ETH from Ethereum to Linea.

Parameter	Value
:	-----
:	-----

Chain ID	59144
Address	<Address contractUrl="https://lineascan.build/address/0xa18152629128738a5c081eb226335FE4B9C95e9" urlId="591440xa18152629128738a5c081eb226335FE4B9C95e9" urlClass="erc- token-address"/>
Name	Chainlink Token on Linea Mainnet
Symbol	LINK
Decimals	18
Network status	linea.statuspage.io

zkSync

zkSync Era mainnet

ETH is used to pay for transactions on zkSync Era Mainnet. Use the recommended zkSync Bridges to transfer ETH from Ethereum to zkSync.

Parameter	Value
:	-----
:	-----

Chain ID	324
Address	<Address contractUrl="https://explorer.zksync.io/address/0x52869bae3E091e36b0915941577F2D47d8d8B534" urlId="3240x52869bae3E091e36b0915941577F2D47d8d8B534" urlClass="erc-token-address"/>
Name	Chainlink Token on zkSync Era Mainnet
Symbol	LINK

Decimals	18
Network status	explorer.zksync.io

zkSync Sepolia testnet

Testnet ETH is used to pay for transactions on zkSync Sepolia testnet.

Testnet ETH and LINK are available at faucets.chain.link/zksync-sepolia.

Parameter	Value
:	-----
:	-----

Chain ID	300
Address	<Address contractUrl="https://sepolia.explorer.zksync.io/address/0x23A1aFD896c8c8876AF46aDc38521f4432658d1e" urlId="3000x23A1aFD896c8c8876AF46aDc38521f4432658d1e" urlClass="erc-token-address"/>
Name	Chainlink Token on zkSync Sepolia testnet
Symbol	LINK
Decimals	18
Network status	uptime.com/statuspage/zkSync

Polygon zkEVM

Polygon zkEVM mainnet

ETH is used to pay for transactions on Polygon zkEVM. Use the Polygon zkEVM Bridge to transfer ETH and LINK to Polygon zkEVM.

Parameter	Value
:	-----
:	-----

ETHCHAINID	1101
Address	<Address contractUrl="https://zkevm.polygonscan.com/address/0xdB7A504CF869484dd6aC5FaF925c8386CBF7573D" urlId="11010xdB7A504CF869484dd6aC5FaF925c8386CBF7573D" urlClass="erc-token-address"/>
Name	Chainlink Token on Polygon zkEVM testnet
Symbol	LINK
Decimals	18

Polygon zkEVM Cardona testnet

Testnet ETH is used to pay for transactions on Polygon zkEVM Cardona testnet. Use the Polygon zkEVM Bridge to transfer testnet ETH to Polygon zkEVM testnet.

Testnet ETH and LINK are available at faucets.chain.link/polygon-zkevm-cardona.

Parameter	Value
:	-----
:	-----

ETHCHAINID	2442
Address	<Address contractUrl="https://cardona-zkevm.polygonscan.com/address/0x5576815a38A3706f37bf815b261cCc7cCA77e975" urlId="24420x5576815a38A3706f37bf815b261cCc7cCA77e975" urlClass="erc-token-address"/>
Name	Chainlink Token on Polygon zkEVM Cardona testnet
Symbol	LINK
Decimals	18

Wemix

Wemix mainnet

WEMIX is used to pay for transactions on the Wemix mainnet. To transfer LINK from Ethereum to Wemix, use Transporter.

Parameter	Value
:	-----
:	-----

ETHCHAINID	1111
Address	<Address contractUrl="https://wemixscan.com/address/0x80f1FcdC96B55e459BF52b998aBBE2c364935d69" urlId="11110x80f1FcdC96B55e459BF52b998aBBE2c364935d69" urlClass="erc-token-address"/>
Name	Chainlink Token on Wemix mainnet
Symbol	LINK
Decimals	18

Wemix testnet

Testnet WEMIX is used to pay for transactions on the Wemix testnet.

Testnet WEMIX and LINK are available at faucets.chain.link/wemix-testnet.
Testnet WEMIX is also available from the WEMIX Faucet.

Parameter	Value
:	-----
:	-----

ETHCHAINID	1112
Address	<Address contractUrl="https://testnet.wemixscan.com/address/0x3580c7a817ccd41f7e02143bfa411d4eeae78093" urlId="11120x3580c7a817ccd41f7e02143bfa411d4eeae78093"

```
urlClass="erc-token-address"/> |
| Name | Chainlink Token on Wemix testnet
|
| Symbol | LINK
|
| Decimals | 18
|
```

Kroma

Kroma mainnet

ETH is used to pay for transactions on the Kroma mainnet.

```
| Parameter | Value
|
| :-----
| :-----
|-----
|----- |
| ETHCHAINID | 255
|
| Address | <Address
contractUrl="https://kromascan.com/address/0xC1F6f7622ad37C3f46cDF6F8AA0344ADE80
BF450" urlId="2550xC1F6f7622ad37C3f46cDF6F8AA0344ADE80BF450" urlClass="erc-
token-address"/> |
| Name | Chainlink Token on Kroma mainnet
|
| Symbol | LINK
|
| Decimals | 18
|
```

Kroma Sepolia testnet

Testnet ETH is used to pay for transactions on the Kroma testnet.

Testnet ETH and LINK are available at faucets.chain.link/kroma-testnet.

```
| Parameter | Value
|
| :-----
| :-----
|-----
|----- |
| ETHCHAINID | 2358
|
| Address | <Address
contractUrl="https://sepolia.kromascan.com/address/0xa75cca5b404ec6f4bb6ec4853d1
77fe7057085c8" urlId="23580xa75cca5b404ec6f4bb6ec4853d177fe7057085c8"
urlClass="erc-token-address"/> |
| Name | Chainlink Token on Kroma Sepolia testnet
|
| Symbol | LINK
|
| Decimals | 18
|
```

Mode

Mode mainnet

ETH is used to pay for transactions on the Mode mainnet.

Parameter	Value
:	-----
:	-----

ETHCHAINID	34443
Address	<Address contractUrl="https://explorer.mode.network/address/0x183E3691EfF3524B2315D3703D94F922CbE51F54" urlId="344430x183E3691EfF3524B2315D3703D94F922CbE51F54" urlClass="erc-token-address"/>
Name	Chainlink Token on Mode mainnet
Symbol	LINK
Decimals	18

Mode Sepolia testnet

Testnet ETH is used to pay for transactions on the Mode testnet.

Testnet ETH and LINK are available at faucets.chain.link/mode-sepolia.

Parameter	Value
:	-----
:	-----

ETHCHAINID	919
Address	<Address contractUrl="https://sepolia.explorer.mode.network/address/0x925a4bfE64AE2bFAC8a02b35F78e60C29743755d" urlId="9190x925a4bfE64AE2bFAC8a02b35F78e60C29743755d" urlClass="erc-token-address"/>
Name	Chainlink Token on Mode Sepolia testnet
Symbol	LINK
Decimals	18

Solana

Solana mainnet

SOL is used to pay for transactions on the Solana network.

Parameter	Value
:	-----
:	-----

Chain ID	mainnet
Address	<Address contractUrl="https://solscan.io/account/y9MdSjd9Beg9EFaeQGdMpESFWLNdSfZKQKeYLBfm njJ"/>
Name	Chainlink Token on Solana Mainnet
Symbol	LINK

```
| Decimals          | 9
|
| Network status | status.solana.com/
|
```

network-integration.mdx:

```
---
section: global
date: Last Modified
title: "Integrating EVM Networks With Chainlink Services"
---
```

```
import { Aside } from "@components"
```

Before an EVM blockchain network can integrate with Chainlink, it must meet certain technical requirements. These requirements are critical for Chainlink nodes and Chainlink services to function correctly on a given network.

<Aside type="note" title="Disclaimer">

The standard EVM requirements required to integrate EVM blockchain networks with Chainlink services can vary, are

subject to change, and are provided here for reference purposes only.

Chainlink services may have unique requirements

that are in addition to the requirements discussed herein.

</Aside>

Standard EVM requirements

Solidity global variables and opcode implementation

Solidity global variables and opcode implementation constructs must meet the following requirements in order for Chainlink services to operate correctly and as expected:

- Global variables: Support all global variables as specified in the Solidity Block and Transaction Properties documentation. For example, the following variables must be supported:

- block.blockhash must return the hash of the requested block for the last 256 blocks

- block.number must return the respective chain's block number

- block.chainID must return the current chain ID

- block.timestamp must return the current block timestamp as seconds since unix epoch

- Opcodes: Support all opcodes and expected behaviors from the OpCodes.sol contract

- Precompiles: Must support all precompile contracts and expected behaviors listed in the Solidity Mathematical and Cryptographic Functions documentation

- Contract size: Must support the maximum contract size defined in EIP-170

- Nonce: The transaction nonce must increase as transactions are confirmed and propagated to all nodes in the network.

Finality

Blockchain development teams must ensure that blocks with a commitment level of finalized are actually final. The properties of the finality mechanism, including underlying assumptions and conditions under which finality violations could occur, must be clearly documented and communicated to application developers in the blockchain ecosystem.

Furthermore, this information should be accessible through RPC API tags finalized from the JSON-RPC specification tags described later in this document.

Standardized RPCs with SLAs

Chainlink nodes use RPCs to communicate with the chain and perform soak testing. It is not possible to ensure the functionality of Chainlink services if RPCs are unstable, underperforming, or nonexistent. RPCs must meet the following requirements:

Dedicated RPC node:

- The chain must provide instructions and hardware requirements to set up and run a full node.
- The archive node setup must also be provided and allow queries of blocks from genesis with transaction history and logs.
- The RPC node must enable and allow configurable settings for the following items:
 - Batch calls
 - Log lookbacks
 - HTTPS and WSS connections

RPC providers:

- Three separate independent RPC providers must be available.
- RPC providers must ensure there is no rate limit.
- RPC providers must have a valid SSL certificate.
- During the trailing 30 days, the RPC providers must meet the following RPC performance requirements:
 - Uptime: At least 99.9%
 - Throughput: Support at least 300 calls per second
 - Latency: Less than 250ms
 - Support SLA: For SEV1 issues, provide a Time to Answer (TTA) of at most 1 hour

Support the Ethereum JSON-RPC Specification

The chain must support the Ethereum JSON-RPC Specification. Chainlink services use several methods to operate on the chain and require a specific response format to those calls in line with the JSON RPC standard of Ethereum. If a response does not match this required format, the call fails and the Chainlink node will stop functioning properly.

The following methods are specifically required and must follow the Ethereum RPC API specification:

- GetCode
- Call
- ChainID
- SendTransaction
- SendRawTransaction
- GetTransactionReceipt
- GetTransactionByHash
- EstimateGas
- GasPrice
- GetTransactionCount
- GetLogs
 - Must follow the spec as defined in EIP-1474. The "latest" block number returned by GetBlockByNumber must also be served by GetLogs with logs.
 - Must accept the blockhash param as defined in EIP-234
- GetBalance
- GetBlockByNumber
- GetBlockByHash

The above RPC methods must have the expected request and response params with expected data types and values as described in the Execution-api spec and Ethereum RPC API Spec.

The network must also support the following items:

- Subscription Methods: Websocket JSON-RPC subscription methods
 - ethsubscribe with support for subscription to newHeads and logs
- Tags: The RPC methods must support the finalized, latest, and pending tags where applicable. They must also support natural numbers for blocks.
- Batch Requests: Must support batching of requests for the GetLogs and GetBlockByNumber methods.
- Response size: Any RPC request including the batch requests must be within the allowed size limit of around 173MB.

ethsendRawTransaction error message mapping to Geth client error messages

Chains must provide an error message mapping between their specific implementation to the error messages detailed below.

When the ethsendRawTransaction call fails, Chainlink nodes must be able to recognize these error categories and determine the next appropriate action. If the error categories are different or cannot be mapped correctly, the Chainlink node will stop functioning properly and stop sending transactions to the chain. The following error messages are specifically critical:

Error	Description
-----	-----
NonceTooLow	Returned when the nonce used for the transaction is too low to use. This nonce likely has been already used on the chain previously.
NonceTooHigh	Returned when the nonce used for the transaction is higher than what the chain can use right now.
ReplacementTransactionUnderpriced	Returned when the transaction gas price used is too low. There is another transaction with the same nonce in the queue, with a higher price.
LimitReached	Returned when there are too many outstanding transactions on the node.
TransactionAlreadyInMempool	Returned when this current transaction was already received and stored.
TerminallyUnderpriced	Returned when the transaction's gas price is too low and won't be accepted by the node.
InsufficientEth	Returned when the account doesn't have enough funds to send this transaction.
TxFeeExceedsCap	Returned when the transaction gas fees exceed the configured cap by this node, and won't be accepted.
L2FeeTooLow	Specific for Ethereum L2s only. Returned when the gas fees are too low, When this error occurs the Suggested Gas Price is fetched again transaction is retried.
L2FeeTooHigh	Specific for Ethereum L2s only. Returned when the total fee is too high. When this error occurs the Suggested Gas Price is fetched again transaction is retried.
L2Full	Specific for Ethereum L2s only. The node is too full, and cannot handle more transactions.
TransactionAlreadyMined	Returned when the current transaction was already accepted and mined into a block.

```
|
| Fatal | Return when something is seriously wrong
with the transaction, and the transaction will never be accepted in the current
format. |
```

For examples of how other chains or clients are using these categories, see the `error.go` file in the go-ethereum repo on GitHub.

For chains with zk-proofs, chains must reject transactions that cause zk-proof overflow with a uniquely identifiable error message.

Any other reasons why transactions might be rejected by a node or sequencer other than malformed input/gasLimits must be detailed.

Clarify use of transaction types

For transaction types other than 0x0 - Legacy, 0x1 - Access List, 0x2 - Dynamic, and 0x3 - Blob, networks must clarify how each transaction type is used. Chainlink nodes must know if the chain uses other types for regular transactions with regular gas so it can correctly estimate gas costs.

Multi-signature wallet support

The chain must provide a supported and audited multi-signature wallet implementation with a UI.

Block explorer support

The chain must provide a block explorer and support for contract and verification APIs.

```
# index.mdx:
```

```
---
section: vrf
date: Last Modified
title: "Chainlink VRF"
isMdx: true
isIndex: true
whatsnext: { "Subscription Method": "/vrf/v2-5/overview/subscription" }
metadata:
  title: "Generate Random Numbers for Smart Contracts using Chainlink VRF"
  description: "Learn how to securely generate random numbers for your smart
contract with Chainlink VRF (an RNG). This guide uses Solidity code examples."
---
```

```
import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"
import { Aside } from "@components"
```

```
<Vrf25Common callout="security" />
```

Chainlink VRF (Verifiable Random Function) is a provably fair and verifiable random number generator (RNG) that enables smart contracts to access random values without compromising security or usability. For each request, Chainlink VRF generates one or more random values and cryptographic proof of how those values were determined. The proof is published and verified onchain before any consuming applications can use it. This process ensures that results cannot be tampered with or manipulated by any single entity including oracle operators, miners, users, or smart contract developers.

```
<Aside type="note" title="Migrate to V2.5">
```

Follow the migration guide to learn how VRF has changed in V2.5 and to get example

```
code.
</Aside>
```

Use Chainlink VRF to build reliable smart contracts for any applications that rely on unpredictable outcomes:

- Building blockchain games and NFTs.
- Random assignment of duties and resources. For example, randomly assigning judges to cases.
- Choosing a representative sample for consensus mechanisms.

VRF v2.5 includes all the original benefits of v2 and the following additional benefits:

- Easier upgrades to future versions.
- The option to pay for requests in either LINK or native tokens.

Learn how to migrate to VRF v2.5.

For help with your specific use case, contact us to connect with one of our Solutions Architects. You can also ask questions about Chainlink VRF on Stack Overflow.

Two methods to request randomness

Similarly to VRF v2, VRF v2.5 will offer two methods for requesting randomness:

- Subscription: Create a subscription account and fund its balance with either native tokens or LINK. You can then connect multiple consuming contracts to the subscription account. When the consuming contracts request randomness, the transaction costs are calculated after the randomness requests are fulfilled and the subscription balance is deducted accordingly. This method allows you to fund requests for multiple consumer contracts from a single subscription.
- Direct funding: Consuming contracts directly pay with either native tokens or LINK when they request random values. You must directly fund your consumer contracts and ensure that there are enough funds to pay for randomness requests.

Choosing the correct method

Depending on your use case, one method might be more suitable than another. Consider the following characteristics when you choose a method:

```
{/ prettier-ignore /}
| Subscription method | Direct funding method |
| ----- | ----- |
| Currently available on VRF v2.5 for all supported networks. | Currently available on VRF v2.5 for all supported networks. |
| Suitable for regular requests | Suitable for infrequent one-off requests |
| Supports multiple VRF consuming contracts connected to one subscription account | Each VRF consuming contract directly pays for its requests |
| VRF costs are calculated after requests are fulfilled and then deducted from the subscription balance. Learn how VRF costs are calculated for the subscription method. | VRF costs are estimated and charged at request time, which may make it easier to transfer the cost of VRF to the end user. Learn how VRF costs are calculated for the direct funding method. |
| Reduced gas overhead and more control over the maximum gas price for requests | Higher gas overhead than the subscription method |
| More random values returned per single request. See the maximum random values per request for the V2.5 subscription supported networks. | Fewer random values returned per single request than the subscription method, due to higher overhead. See the maximum random values per request and gas overhead for the V2 direct funding supported networks. |
| You don't have to estimate costs precisely for each request. Ensure that the
```


subscription account has enough funds. | You must estimate transaction costs carefully for each request to ensure the consuming contract has enough funds to pay for the request. |
| Requires a subscription account | No subscription account required |
| VRF costs are billed to your subscription account | No refunds for overpayment after requests are completed |

Supported networks

The contract addresses and gas price limits are different depending on which method you use to get randomness. You can find the configuration, addresses, and limits for each method on the Supported networks page.

To learn when VRF v2.5 becomes available on more networks, follow us on Twitter or sign up for our mailing list.

release-notes.mdx:

```
---
section: vrf
date: Last Modified
title: "Chainlink VRF Release Notes"
isMdx: true
whatsnext: { "Migrate to VRF V2.5": "/vrf/v2-5/migration-from-v2" }
---
```

2024-08-15 - VRF V2.5 on BASE

VRF V2.5 is available on BASE mainnet and BASE Sepolia testnet. See the VRF V2.5 Supported Networks page to get configuration details for both subscription and direct funding.

2024-07-15 - Deprecation announcement

VRF V2 and V1 will be deprecated on November 29, 2024. Please migrate to VRF V2.5 before then.

The docs for VRF V2 are still available and have been moved to the legacy section:

- VRF V2 Subscription method
- VRF V2 Direct funding method

There is also a migration guide available for V1 users migrating to V2.5.

2024-05-24 - Updated VRF V2.5 contracts

The @chainlink/contracts package version 1.1.1 is now available. It includes the updated wrapper and interface contracts for VRF 2.5 direct funding, which had not been included in the @chainlink/contracts package version 1.1.0. The DirectFundingConsumer.sol example contract has been updated to reflect this.

The @chainlink/contracts also includes an updated function signature for fulfillRandomWords in the VRFConsumerBaseV2Plus contract, which applies only to subscription users. This function signature has not changed in the VRFV2PlusWrapperConsumerBase, so this does not affect direct funding users.

When using package version 1.1.1 and later, update your fulfillRandomWords function signature to match the VRFConsumerBaseV2Plus contract, which has changed to:

```
function fulfillRandomWords(uint256 requestId, uint256[] calldata randomWords)
```

In the @chainlink/contracts package version 1.1.0 and earlier, the randomWords parameter has a memory storage location.

2024-04-29 - VRF V2.5 released on Ethereum, BNB Chain, Polygon, Avalanche and Arbitrum

VRF V2.5 is available on Ethereum, BNB Chain, Polygon, Avalanche and Arbitrum mainnets and testnets.

The new version of Chainlink VRF implements the following changes:

- Support for native gas token billing
- Easy 1-click migration to future new versions
- New billing model, where the premium is a percentage of the gas costs of the VRF callback instead of a flat fee
- Gas optimizations

Learn how to migrate to VRF V2.5.

2024-04-13 - Polygon testnet support changed

The Mumbai network has stopped producing blocks, so example code will not function on this network. Check again soon for updates about future testnet support on Polygon.

2024-03-29 - Fantom support changed

Creating new Fantom subscriptions in the VRF Subscription Manager is no longer supported. Existing Fantom subscriptions are still supported.

2023-11-17 - Arbitrum testnet changed

Arbitrum Goerli support ends as of November 18, 2023. Support for Arbitrum Sepolia is available for both subscription and direct funding.

2023-10-02 - VRF Quickstarts and Resources

The Developer Hub has been released. It helps you find resources related to web3 use cases like NFTs and gaming. The Developer Hub includes a comprehensive VRF Resources page that shows a collection of Quickstarts, guides, tutorials, videos, blog posts, courses, documentation, and case studies related to VRF.

A new set of Quickstarts has been released. See all the Quickstarts that involve VRF.

2023-10-23 - Sepolia gas lane increase

For VRF V2 subscription, the Sepolia gas lane has increased from 30 gwei to 150 gwei. The key hash has otherwise remained the same.

2023-10-02 - VRF Quickstarts and Resources

The Developer Hub has been released. It helps you find resources related to web3 use cases like NFTs and gaming. The Developer Hub includes a comprehensive VRF Resources page that shows a collection of Quickstarts, guides, tutorials, videos, blog posts, courses, documentation, and case studies related to VRF.

A new set of Quickstarts has been released. See all the Quickstarts that involve VRF.

2023-07-26 - VRF Cost Calculator

A VRF cost calculator has been added to the Estimating Costs page. Use this calculator to estimate costs for both subscription and direct funding.

2023-06-14 - Arbitrum support and docs expanded

Arbitrum mainnet and Arbitrum Goerli are supported on VRF V2 direct funding. Detailed cost explanations for Arbitrum and a cost estimation code example are available on the Estimating Costs page.

2023-05-19 - Support added for Arbitrum

Arbitrum mainnet and Arbitrum Goerli are supported on VRF V2 subscription.

2023-04-20 - Supported network removed

Klaytn and Klaytn Baobab are no longer supported networks on VRF.

See the currently supported networks for subscription and direct funding.

2023-04-19 - VRF Subscription Manager updated

The VRF Subscription Manager has a new Actions menu that displays actions you can take on a VRF subscription, including funding, cancellation, and adding an email address.

2023-04-12 - Estimating costs page added

Billing and cost information for VRF V2 subscription and direct funding has been consolidated into one Estimating Costs page. Static cost breakdown examples are available for both funding methods.

2023-04-05 - VRF V2 mock contracts added

Mock contracts for local testing are available for VRF V2 subscription and direct funding:

- Test VRF V2 subscription locally
- Test VRF V2 direct funding locally

2022-02-16 - VRF V2 is GA

VRF V2 is generally available with new sample contracts for V2.

2021-12-14 - VRF V2 launched

VRF V2 is available along with guides to help you migrate from V1 to V2:

- Subscription: Migrating from V1 to V2
- Direct funding: Migrating from V1 to V2

2020-10-22 - VRF V1 is available

VRF V1 is available on Ethereum mainnet.

api-reference.mdx:

section: legacy

date: Last Modified

title: "Chainlink VRF API Reference [v1]"

metadata:

title: "Chainlink VRF API Reference"

description: "API reference for VRFConsumerBase."

```
import VrfCommon from "@features/vrf/v1/common/VrfCommon.astro"
import { Aside } from "@components"
```

```
<Aside type="caution" title="Migrate to VRF V2.5">
  VRF V2.5 replaces both VRF V1 and VRF V2 on November 29, 2024. Migrate to VRF
  V2.5.
</Aside>
```

```
<VrfCommon />
```

API reference for VRFConsumerBase.

Index

Constructors

Name	Description
constructor	Initialize your consuming contract.

Functions

Name	Description
requestRandomness	Make a request to the VRFCoordinator.
fulfillRandomness	Called by VRFCoordinator when it receives a valid VRF proof.

Constructor

Initialize your consuming contract.

```
{/ prettier-ignore /}
solidity
constructor(address vrfCoordinator, address link)
```

- vrfCoordinator: Address of the Chainlink VRF Coordinator. See Chainlink VRF Addresses for details.

- link: Address of the LINK token. See LINK Token Addresses for details.

> Note: seed has recently been deprecated.

Functions

requestRandomness

Make a request to the VRF coordinator.

```
{/ prettier-ignore /}
solidity
function requestRandomness(bytes32 keyHash, uint256 fee)
  public returns (bytes32 requestId)
```

- keyHash: The public key against which randomness is generated. See Chainlink VRF supported networks for details.
- fee: The fee, in LINK, for the request. Specified by the oracle.
- RETURN: The ID unique to a single request.

fulfillRandomness

Called by VRFCoordinator when it receives a valid VRF proof. Override this function to act upon the random number generated by Chainlink VRF.

```
{/ prettier-ignore /}
solidity
function fulfillRandomness(bytes32 requestId, uint256 randomness)
    internal virtual;
```

- requestId: The ID initially returned by requestRandomness.
- randomness: The random number generated by Chainlink VRF.

Reference

Maximizing security

Chainlink VRF provides powerful security guarantees and is easy to integrate. However, smart contract security is a nuanced topic. You can read about the top security considerations for VRF.

best-practices.mdx:

```
---
section: legacy
date: Last Modified
title: "VRF Best Practices [v1]"
metadata:
  title: "Chainlink VRF API Reference"
  description: "Best practices for using Chainlink VRF."
---

import VrfCommon from "@features/vrf/v1/common/VrfCommon.astro"
import { Aside, CodeSample } from "@components"

<Aside type="caution" title="Migrate to VRF V2.5">
  VRF V2.5 replaces both VRF V1 and VRF V2 on November 29, 2024. Migrate to VRF
  V2.5.
</Aside>

<VrfCommon />
```

Below are the best practices for using Chainlink VRF.

Getting a random number within a range

If you need to generate a random number within a given range, you use modulo to define the limits of your range. Below you can see how to get a random number between 1 and 50.

```
{/ prettier-ignore /}
solidity
uint256 public randomResult;

function fulfillRandomness(bytes32 requestId, uint256 randomness) internal
override {
    randomResult = (randomness % 50) + 1;
```

```
}
```

Getting multiple random numbers

If you want to get multiple random numbers from a single VRF response, you should create an array where the randomValue is your original returned VRF number and n is the desired number of random numbers.

```
{/ prettier-ignore /}  
solidity  
function expand(uint256 randomValue, uint256 n) public pure returns (uint256[]  
memory expandedValues) {  
    expandedValues = new uint256[](n);  
    for (uint256 i = 0; i < n; i++) {  
        expandedValues[i] = uint256(keccak256(abi.encode(randomValue, i)));  
    }  
    return expandedValues;  
}
```

Having multiple VRF requests in flight

If you want to have multiple VRF requests in flight, you might want to create a mapping between the requestId and the address of the requester.

```
{/ prettier-ignore /}  
solidity  
mapping(bytes32 => address) public requestIdToAddress;  
  
function getRandomNumber() public returns (bytes32 requestId) {  
    require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK - fill  
contract with faucet");  
    bytes32 requestId = requestRandomness(keyHash, fee);  
    requestIdToAddress[requestId] = msg.sender;  
}  
  
function fulfillRandomness(bytes32 requestId, uint256 randomness) internal  
override {  
    address requestAddress = requestIdToAddress[requestId];  
}
```

If you want to keep order when a request was made, you might want to use a mapping of requestId to the index/order of this request.

```
{/ prettier-ignore /}  
solidity  
mapping(bytes32 => uint256) public requestIdToRequestNumberIndex;  
uint256 public requestCounter;  
  
function getRandomNumber() public returns (bytes32 requestId) {  
    require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK - fill  
contract with faucet");  
    bytes32 requestId = requestRandomness(keyHash, fee);  
    requestIdToRequestNumberIndex[requestId] = requestCounter;  
    requestCounter += 1;  
}  
  
function fulfillRandomness(bytes32 requestId, uint256 randomness) internal  
override {  
    uint256 requestNumber = requestIdToRequestNumberIndex[requestId];  
}
```

If you want to keep generated random numbers of several VRF requests, you might want to use a mapping of requestId to the returned random number.

```
{/ prettier-ignore /}
solidity
mapping(bytes32 => uint256) public requestIdToRandomNumber;

function getRandomNumber() public returns (bytes32 requestId) {
    require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK - fill
contract with faucet");
    return requestRandomness(keyHash, fee);
}

function fulfillRandomness(bytes32 requestId, uint256 randomness) internal
override {
    requestIdToRandomNumber[requestId] = randomness;
}
```

Feel free to use whatever data structure you prefer.

introduction.mdx:

```
---
section: legacy
date: Last Modified
title: "Introduction to Chainlink VRF [v1]"
whatsnext:
  {
    "Get a Random Number": "/vrf/v1/examples/get-a-random-number",
    "API Reference": "/vrf/v1/api-reference",
    "Supported Networks": "/vrf/v1/supported-networks",
  }
metadata:
  title: "Generate Random Numbers for Smart Contracts using Chainlink VRF"
  description: "Learn how to securely generate random numbers for your smart
contract with Chainlink VRF (an RNG). This guide uses Solidity code examples."
---
```

```
import VrfCommon from "@features/vrf/v1/common/VrfCommon.astro"
import { Aside, CodeSample } from "@components"

<Aside type="caution" title="Migrate to VRF V2.5">
  VRF V2.5 replaces both VRF V1 and VRF V2 on November 29, 2024. Migrate to VRF
V2.5.
</Aside>

<VrfCommon />
```

Generate Random Numbers in your Smart Contracts

Chainlink VRF (Verifiable Random Function) is a provably-fair and verifiable source of randomness designed for smart contracts. Smart contract developers can use Chainlink VRF as a tamper-proof random number generator (RNG) to build reliable smart contracts for any applications which rely on unpredictable outcomes:

- Blockchain games and NFTs
- Random assignment of duties and resources (e.g. randomly assigning judges to cases)
- Choosing a representative sample for consensus mechanisms

Learn how to write smart contracts that consume random numbers: [Get a Random Number](#).

onchain Verification of Randomness

Chainlink VRF enables smart contracts to access randomness without compromising on security or usability. With every new request for randomness, Chainlink VRF generates a random number and cryptographic proof of how that number was determined. The proof is published and verified onchain before it can be used by any consuming applications. This process ensures that the results cannot be tampered with nor manipulated by anyone, including oracle operators, miners, users and even smart contract developers.

[Read more about Chainlink VRF in our announcement post.](#)

```
# security.mdx:
```

```
---
section: legacy
date: Last Modified
title: "VRF Security Considerations [v1]"
---
```

```
import VrfCommon from "@features/vrf/v1/common/VrfCommon.astro"
import { Aside, CodeSample } from "@components"
```

```
<Aside type="caution" title="Migrate to VRF V2.5">
  VRF V2.5 replaces both VRF V1 and VRF V2 on November 29, 2024. Migrate to VRF
  V2.5.
</Aside>
```

```
<VrfCommon />
```

Gaining access to high quality randomness onchain requires a solution like Chainlink's VRF, but it also requires you to understand some of the ways that randomness generation can be manipulated by miners/validators. Here are some of the top security considerations you should review in your project.

- Use `requestId` to match randomness requests with their fulfillment in order
- Choose a safe block confirmation time, which will vary between blockchains
- Do not allow re-requesting or cancellation of randomness
- Don't accept bids/bets/inputs after you have made a randomness request
- `fulfillRandomness` must not revert
- Use `VRFConsumerBase` in your contract, to interact with the VRF service

Use `requestId` to match randomness requests with their fulfillment in order

If your contract could have multiple VRF requests in flight simultaneously, you must ensure that the order in which the VRF fulfillments arrive cannot be used to manipulate your contract's user-significant behavior.

Blockchain miners/validators can control the order in which your requests appear onchain, and hence the order in which your contract responds to them.

For example, if you made randomness requests A, B, C in short succession, there is no guarantee that the associated randomness fulfillments will also be in order A, B, C. The randomness fulfillments might just as well arrive at your contract in order C, A, B or any other order.

We recommend using the `requestID` to match randomness requests with their corresponding fulfillments.

Choose a safe block confirmation time, which will vary between blockchains

In principle, miners and validators of your underlying blockchain could rewrite the chain's history to put a randomness request from your contract into a different block, which would result in a different VRF output. Note that this does not enable a miner to determine the random value in advance. It only enables them to get a fresh random value that may or not be to their advantage. By way of analogy, they can only re-roll the dice, not predetermine or predict which side it will land on.

You must choose an appropriate confirmation time for the randomness requests you make (i.e. how many blocks the VRF service waits before writing a fulfillment to the chain) to make such rewrite attacks unprofitable in the context of your application and its value-at-risk.

On proof-of-stake blockchains (e.g. BSC, Polygon), what block confirmation time is considered secure depends on the specifics of their consensus mechanism and whether you're willing to trust any underlying assumptions of (partial) honesty of validators.

For further details, take a look at the consensus documentation for the chain you want to use:

- Ethereum Consensus Mechanisms
- BNB Chain Consensus Docs
- Polygon Consensus Docs

Understanding the blockchains you build your application on is very important. You should take time to understand chain reorganization which will also result in a different VRF output, which could be exploited.

Do not allow re-requesting or cancellation of randomness

Any re-request or cancellation of randomness is an incorrect use of VRF. dApps that implement the ability to cancel or re-request randomness for specific commitments must consider the additional attack vectors created by this capability. For example, you must prevent the ability for any party to discard unfavorable randomness.

Don't accept bids/bets/inputs after you have made a randomness request

Consider the example of a contract that mints a random NFT in response to a users' actions.

The contract should:

1. record whatever actions of the user may affect the generated NFT
1. stop accepting further user actions that may affect the generated NFT and issue a randomness request
1. on randomness fulfillment, mint the NFT

Generally speaking, whenever an outcome in your contract depends on some user-supplied inputs and randomness, the contract should not accept any additional user-supplied inputs once the randomness request has been issued.

Otherwise, the cryptoeconomic security properties may be violated by an attacker that can rewrite the chain.

fulfillRandomness must not revert

If your fulfillRandomness implementation reverts, the VRF service will not attempt to call it a second time. Make sure your contract logic does not revert. Consider simply storing the randomness and taking more complex follow-on actions in separate contract calls made by you or your users.

Use VRFConsumerBase in your contract, to interact with the VRF service

VRFConsumerBase tracks important state which needs to be synchronized with the VRFCoordinator state. Some users fold VRFConsumerBase into their own contracts, but this means taking on significant extra complexity, so we advise against doing so.

Along the same lines, don't override rawFulfillRandomness.

```
# supported-networks.mdx:
```

```
---
section: legacy
date: Last Modified
title: "Chainlink VRF Supported Networks [v1]"
metadata:
  title: "Chainlink VRF v1 Supported Networks"
---
```

```
import VrfCommon from "@features/vrf/v1/common/VrfCommon.astro"
import ResourcesCallout from
"@features/resources/callouts/ResourcesCallout.astro"
import { Aside } from "@components"
```

```
<Aside type="caution" title="Migrate to VRF V2.5">
  VRF V2.5 replaces both VRF V1 and VRF V2 on November 29, 2024. Migrate to VRF
V2.5.
</Aside>
```

Chainlink VRF allows you to integrate provably-fair and verifiably random data in your smart contract.

For implementation details, read [Introduction to Chainlink VRF](#).

```
<ResourcesCallout callout="bridgeRisks" />
```

Polygon Mainnet

```
<Aside type="tip" title="Important">
  The LINK provided by the Polygon Bridge is not ERC-677 compatible, so
  cannot be used with Chainlink oracles. However, it can be converted to the
official LINK token on Polygon using
  Chainlink's PegSwap service
</Aside>
```

Item	Value

LINK Token	0xb0897686c545045aFc77CF20eC7A532E3120E0F1
VRF Coordinator	0x3d2341ADb2D31f1c5530cDC622016af293177AE0
Key Hash	0xf86195cf7690c55907b2b611ebb7343a6f649bff128701cc542f0569e2c549da
Fee	0.0001 LINK

```
<Aside type="note" title="VRF Response Times on Polygon">
  VRF responses are generated after 10 block confirmations on Polygon by
default.
</Aside>
```

BNB Chain Mainnet

<Aside type="note" title="Important">

The LINK provided by the BNB Chain Bridge is not ERC-677 compatible, so cannot be used with Chainlink oracles. However, it can be converted to the official LINK token on BNB Chain using Chainlink's PegSwap service.

</Aside>

Item	Value

LINK Token	0x404460C6A5EdE2D891e8297795264fDe62ADBB75
VRF Coordinator	0x747973a5A2a4Ae1D3a8fDF5479f1514F65Db9C31
Key Hash	0xc251acd21ec4fb7f31bb8868288bfdbaeb4fbfec2df3735ddbd4f7dc8d60103c
Fee	0.2 LINK - initial fees on BNB Chain are meant to cover the highest gas cost prices.

BNB Chain Testnet

<Aside type="note" title="BNB Chain Faucet">

Testnet LINK is available from <https://faucets.chain.link/bnb-chain-testnet>

</Aside>

Item	Value

LINK	0x84b9B910527Ad5C03A9Ca831909E21e236EA7b06
VRF Coordinator	0xa555fC018435bef5A13C6c6870a9d4C11DEC329C
Key Hash	0xcaf3c3727e033261d383b315559476f48034c13b18f8cafed4d871abe5049186
Fee	0.1 LINK

Ethereum Mainnet

Item	Value

LINK Token	0x514910771AF9Ca656af840dff83E8264EcF986CA
VRF Coordinator	0xf0d54349aDdcf704F77AE15b96510dEA15cb7952
Key Hash	0xAA77729D3466CA35AE8D28B3BBAC7CC36A5031EFDC430821C02BC31A238AF445
Fee	2 LINK - initial fees on Ethereum are meant to cover the highest gas cost prices.

get-a-random-number.mdx:

```
---
section: legacy
date: Last Modified
title: "Get a Random Number [v1]"
whatsnext: { "Migrate to VRF V2.5": "/vrf/v2-5/migration-from-v1" }
metadata:
  description: "How to generate a random number inside a smart contract using
Chainlink VRF."
---
```

```
import VrfCommon from "@features/vrf/v1/common/VrfCommon.astro"
import { Aside, CodeSample } from "@components"

<Aside type="caution" title="Migrate to VRF V2.5">
  VRF V2.5 replaces both VRF V1 and VRF V2 on November 29, 2024. Migrate to VRF
V2.5.
</Aside>
```

```
<VrfCommon />
```

This page explains how to get a random number inside a smart contract using Chainlink VRF.

Random Number Consumer

Chainlink VRF follows the Request & Receive Data cycle. To consume randomness, your contract should inherit from `VRFConsumerBase` 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.

The contract should own enough LINK to pay the specified fee. The beginner walkthrough explains how to fund your contract.

Note, the below values have to be configured correctly for VRF requests to work. You can find the respective values for your network in the VRF Contracts page.

- LINK Token - LINK token address on the corresponding network (Ethereum, Polygon, BSC, etc)
- VRF Coordinator - address of the Chainlink VRF Coordinator
- Key Hash - public key against which randomness is generated
- Fee - fee required to fulfill a VRF request

```
<Aside type="tip" title="Security Considerations">
  Be sure to look your contract over with these security considerations in mind!
</Aside>
```

```
<Aside type="note" title="Remember to fund your contract with LINK!">
  Requesting randomness will fail unless your deployed contract has enough LINK
to pay for it. Learn how to Acquire
  testnet LINK and Fund your contract.
</Aside>
```

```
<CodeSample src="/samples/VRF/RandomNumberConsumer.sol" />
```

```
<Aside type="note" title="Maximum Gas for Callback">
  If your fulfillRandomness function uses more than 200k gas, the transaction
will fail.
</Aside>
```

Getting More Randomness

If you are looking for how to turn a single result into multiple random numbers, check out our guide on Randomness Expansion.

Network Congestion and Responsiveness

Network congestion can occur on all blockchains from time to time, which may result in transactions taking longer to get included in a block. During times of network congestion, the VRF service will continue responding to randomness requests, but fulfillment response times will correspondingly increase based on the level of congestion. It is important you account for this in your use case and set expectations accordingly.

best-practices.mdx:

```
---
section: legacy
date: Last Modified
title: "VRF V2 Best Practices"
metadata:
  title: "Chainlink VRF API Reference"
  description: "Best practices for using Chainlink VRF."
---

import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
import { Aside, CodeSample } from "@components"

<Aside type="tip" title="VRF V2.5 Best Practices">
  Refer to the VRF V2.5 Best Practices page for VRF best practices with updated
  VRF 2.5 code
  examples.
</Aside>

<VrfCommon callout="common" />
```

These are example best practices for using Chainlink VRF. To explore more applications of VRF, refer to our blog.

Getting a random number within a range

If you need to generate a random number within a given range, use modulo to define the limits of your range. Below you can see how to get a random number in a range from 1 to 50.

```
{/ prettier-ignore /}
solidity
function fulfillRandomWords(
  uint256, / requestId /
  uint256[] memory randomWords
) internal override {
  // Assuming only one random word was requested.
  srandomRange = (randomWords[0] % 50) + 1;
}
```

Getting multiple random values

If you want to get multiple random values from a single VRF request, you can request this directly with the numWords argument:

- If you are using the VRF v2 subscription method, see the Get a Random Number guide for an example where one request returns multiple random values.
- If you are using the VRF v2 direct funding method, see the Get a Random Number

guide for an example where one request returns multiple random values.

Processing simultaneous VRF requests

If you want to have multiple VRF requests processing simultaneously, create a mapping between requestId and the response. You might also create a mapping between the requestId and the address of the requester to track which address made each request.

```

{
  // prettier-ignore
}
solidity
mapping(uint256 => uint256[]) public srequestIdToRandomWords;
mapping(uint256 => address) public srequestIdToAddress;
uint256 public srequestId;

function requestRandomWords() external onlyOwner returns (uint256) {
    uint256 requestId = COORDINATOR.requestRandomWords(
        keyHash,
        ssubscriptionId,
        requestConfirmations,
        callbackGasLimit,
        numWords
    );
    srequestIdToAddress[requestId] = msg.sender;

    // Store the latest requestId for this example.
    srequestId = requestId;

    // Return the requestId to the requester.
    return requestId;
}

function fulfillRandomWords(
    uint256 requestId,
    uint256[] memory randomWords
) internal override {
    // You can return the value to the requester,
    // but this example simply stores it.
    srequestIdToRandomWords[requestId] = randomWords;
}

```

You could also map the requestId to an index to keep track of the order in which a request was made.

```

{
  // prettier-ignore
}
solidity
mapping(uint256 => uint256) srequestIdToRequestIndex;
mapping(uint256 => uint256[]) public srequestIndexToRandomWords;
uint256 public requestCounter;

function requestRandomWords() external onlyOwner {
    uint256 requestId = COORDINATOR.requestRandomWords(
        keyHash,
        ssubscriptionId,
        requestConfirmations,
        callbackGasLimit,
        numWords
    );
    srequestIdToRequestIndex[requestId] = requestCounter;
    requestCounter += 1;
}

function fulfillRandomWords(

```

```

    uint256 requestId,
    uint256[] memory randomWords
) internal override {
    uint256 requestNumber = srequestIdToRequestIndex[requestId];
    srequestIndexToRandomWords[requestNumber] = randomWords;
}

```

Processing VRF responses through different execution paths

If you want to process VRF responses depending on predetermined conditions, you can create an enum. When requesting for randomness, map each requestId to an enum. This way, you can handle different execution paths in fulfillRandomWords. See the following example:

```
<CodeSample src="samples/VRF/VRFv2MultiplePaths.sol" />
```

```
# estimating-costs.mdx:
```

```

---
section: legacy
date: Last Modified
title: "VRF Billing"
whatsnext:
  {
    "Get a Random Number": "/vrf/v2/direct-funding/examples/get-a-random-
number",
    "Supported Networks": "/vrf/v2/direct-funding/supported-networks",
  }
metadata:
  title: "Estimating costs for Chainlink VRF v2"
  description: "Learn how to estimate costs for Chainlink VRF v2."
isMdx: true
---

```

```

import { Tabs, TabsContent } from "@components/Tabs"
import { Aside, CodeSample } from "@components"
import DropDown from "@features/vrf/v2/components/Dropdown.astro"

```

This guide explains how to estimate VRF V2 costs for both the subscription and direct funding methods.

```

<Aside type="tip" title="VRF V2.5 billing information">
  VRF V2.5 adds the option to pay for VRF requests with native tokens as well as
  LINK. Learn more about VRF V2.5
  billing.
</Aside>

```

VRF v2 cost calculator

```
<DropDown />
```

Understanding transaction costs

```

<TabsContent sharedStore="vrfMethod" client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">

```

For Chainlink VRF v2 to fulfill your requests, you must maintain a sufficient amount of LINK in your subscription balance. Gas cost calculation includes the following variables:

- Gas price: The current gas price, which fluctuates depending on network conditions.
- Callback gas: The amount of gas used for the callback request that returns your requested random values.
- Verification gas: The amount of gas used to verify randomness onchain.

The gas price depends on current network conditions. The callback gas depends on your callback function, and the number of random values in your request. The cost of each request is final only after the transaction is complete, but you define the limits you are willing to spend for the request with the following variables:

- Gas lane: The maximum gas price you are willing to pay for a request in wei. Define this limit by specifying the appropriate keyHash in your request. The limits of each gas lane are important for handling gas price spikes when Chainlink VRF bumps the gas price to fulfill your request quickly.
- Callback gas limit: Specifies the maximum amount of gas you are willing to spend on the callback request. Define this limit by specifying the callbackGasLimit value in your request.

```
</Fragment>
<Fragment slot="panel.2">
```

For Chainlink VRF v2 to fulfill your requests, you must have a sufficient amount of LINK in your consuming contract. Gas cost calculation includes the following variables:

- Gas price: The current gas price, which fluctuates depending on network conditions.
- Callback gas: The amount of gas used for the callback request that returns your requested random values. The callback gas depends on your callback function and the number of random values in your request. Set the callback gas limit to specify the maximum amount of gas you are willing to spend on the callback request.
- Verification gas: The amount of gas used to verify randomness onchain.
- Wrapper overhead gas: The amount of gas used by the VRF Wrapper contract. See the Request and Receive Data section for details about the VRF v2 Wrapper contract design.

Because the consuming contract directly pays the LINK for the request, the cost is calculated during the request and not during the callback when the randomness is fulfilled. Test your callback function to learn how to correctly estimate the callback gas limit.

- If the gas limit is underestimated, the callback fails and the consuming contract is still charged for the work done to generate the requested random values.
- If the gas limit is overestimated, the callback function will be executed but your contract is not refunded for the excess gas amount that you paid.

Make sure that your consuming contracts are funded with enough LINK tokens to cover the transaction costs. If the consuming contract doesn't have enough LINK tokens, your request will revert.

```
</Fragment>
</TabsContent>
```

Estimate gas costs


```

<TabsContent sharedStore="vrfMethod" client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">

```

You need to pre-fund your subscription enough to meet the minimum subscription balance in order to have a buffer against gas volatility.

After the request is complete, the final gas cost is recorded based on how much gas is used for the verification and callback. The actual cost of the request is deducted from your subscription balance.

The total gas cost in wei for your request uses the following formula:

$$(\text{Gas price} \times (\text{Verification gas} + \text{Callback gas})) = \text{total gas cost}$$

The total gas cost is converted to LINK using the ETH/LINK data feed. In the unlikely event that the data feed is unavailable, the VRF coordinator uses the fallbackWeiPerUnitLink value for the conversion instead. The fallbackWeiPerUnitLink value is defined in the coordinator contract for your selected network.

```

</Fragment>
<Fragment slot="panel.2">

```

The final gas cost to fulfill randomness is estimated based on how much gas is expected for the verification and callback. The total gas cost in wei uses the following formula:

$$(\text{Gas price} \times (\text{Verification gas} + \text{Callback gas limit} + \text{Wrapper gas overhead})) = \text{total gas cost}$$

The total gas cost is converted to LINK using the ETH/LINK data feed. In the unlikely event that the data feed is unavailable, the VRF Wrapper uses the fallbackWeiPerUnitLink value for the conversion instead. The fallbackWeiPerUnitLink value is defined in the VRF v2 Wrapper contract for your selected network.

The maximum allowed callbackGasLimit value for your requests is defined in the Coordinator contract supported networks page. Because the VRF v2 Wrapper adds a gas overhead, your callbackGasLimit must not exceed maxGasLimit - wrapperGasOverhead.

```

</Fragment>
</TabsContent>

```

Add LINK premium

```

<TabsContent sharedStore="vrfMethod" client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">

```

The LINK premium is added to the total gas cost. The premium is defined in the coordinator contract with the fulfillmentFlatFeeLinkPPMTier1 parameter in millionths of LINK.

$(\text{total gas cost} + \text{LINK premium}) = \text{total request cost}$

The total request cost is charged to your subscription balance.

```
</Fragment>
<Fragment slot="panel.2">
```

A LINK premium is then added to the total gas cost. The premium is divided in two parts:

- Wrapper premium: The premium percentage. You can find the percentage for your network in the Supported networks page.
- Coordinator premium: A flat fee. This premium is defined in the fulfillmentFlatFeeLinkPPMTier1 parameter in millionths of LINK. You can find the flat fee of the coordinator for your network in the Supported networks page.

$(\text{Coordinator premium} + (\text{total gas cost} \times (1 + \text{Wrapper premium percentage}))) = \text{total request cost}$

```
</Fragment>
</TabsContent>
```

Ethereum example

```
<TabsContent sharedStore="vrfMethod" client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
```

This is an example calculation of a VRF request on the Ethereum network. The values for other supported networks are available on the Supported Networks page.

Estimate minimum subscription balance

You need to have the minimum subscription balance for your requests to be processed. This provides a buffer in case gas prices go higher when processing the request. The actual cost of the request is usually lower than the minimum subscription balance.

Parameter	Value
Gas lane	500 gwei
Callback gas limit	100000
Max verification gas	200000
LINK premium	0.25 LINK

1. Calculate the total gas cost, using the maximum possible gas price for the selected gas lane, the estimated maximum verification gas, and the full callback gas limit:

Gas cost calculation	Total gas cost
Gas price x (Verification gas + Callback gas)	
500 gwei x (200000 + 100000)	150000000 gwei (0.15 ETH)

1. Convert the gas cost to LINK using the LINK/ETH feed.
For this example, assume the feed returns a conversion value of 1 0.004 ETH per 1 LINK.

ETH to LINK cost conversion	Total gas cost (LINK)
-----	-----
0.15 ETH / 0.004 ETH/LINK	37.5 LINK

1. Add the LINK premium to get the total maximum cost of a request:

Adding LINK premium	Maximum request cost (LINK)
-----	-----
Total gas cost (LINK) + LINK premium	
37.5 LINK + 0.25 LINK	37.75 LINK

This example request requires a minimum subscription balance of 37.75 LINK. Check the Max Cost in the Subscription Manager to view the minimum subscription balance for all your contracts. When your request is processed, the actual cost of the request is deducted from your subscription balance.

Estimate VRF request cost

This example reflects an estimate of how much a VRF request costs. Check Etherscan for current gas prices.

Parameter	Value
-----	-----
Actual gas price	50 gwei
Callback gas used	95000
Verification gas used	115000
LINK premium	0.25 LINK

1. Calculate the total gas cost:

Gas cost calculation	Total gas cost
-----	-----
Gas price x (Verification gas + Callback gas)	
50 gwei x (115000 + 95000)	10500000 gwei (0.0105 ETH)

1. Convert the gas cost to LINK using the LINK/ETH feed.

For this example, assume the feed returns a conversion value of 0.004 ETH per 1 LINK.

ETH to LINK cost conversion	Total gas cost (LINK)
-----	-----
0.0105 ETH / 0.004 ETH/LINK	2.625 LINK

1. Add the LINK premium to get the total cost of a request:

Adding LINK premium	Total request cost (LINK)
-----	-----
Total gas cost (LINK) + LINK premium	
2.625 LINK + 0.25 LINK	2.875 LINK

This example request would cost 2.875 LINK, which is deducted from your subscription balance.

</Fragment>

<Fragment slot="panel.2">

This is an example calculation of a VRF request on the Ethereum network. The values for other supported networks are available on the Supported Networks page.

Parameter	Value
Gas price	50 gwei
Callback gas limit	100000
Coordinator gas overhead (Verification gas)	90000
Wrapper gas overhead	40000
Coordinator premium	0.25 LINK
Wrapper premium percentage	0

Steps

1. Calculate the total gas cost:

Gas cost calculation
Total gas cost

Gas price x (Verification gas + Callback gas limit + Wrapper gas overhead)
50 gwei x (90000 + 100000 + 40000)
11500000 gwei (0.0115 ETH)

1. Convert the gas cost to LINK using the LINK/ETH feed.
For this example, assume the feed returns a conversion value of $\hat{1}$ 0.004 ETH per 1 LINK.

ETH to LINK cost conversion	Total gas cost (LINK)
-----	-----
0.0115 ETH / 0.004 ETH/LINK	2.875 LINK

1. Add the LINK premium to get the total cost of a request:

Adding LINK premium
Total request cost (LINK)

Coordinator premium + (Total gas cost x (1 + Wrapper premium percentage))
0.25 LINK + (2.875 x (1 + 0))
3.125 LINK

This example request would cost 3.125 LINK.

</Fragment>
</TabsContent>

Arbitrum

The total transaction costs for using Arbitrum involve both L2 gas costs and L1 costs. Arbitrum transactions are posted in batches to L1 Ethereum, which incurs an L1 cost. For an individual transaction, the total cost includes part of the L1 cost incurred to post the batch that included the transaction.

To learn how to estimate gas costs for Arbitrum, refer to the Arbitrum gas estimation tutorial and the full Arbitrum gas estimation script using their SDK. There is also a version of Arbitrum's gas estimation script extended to include VRF calculations.

Estimating Arbitrum gas costs with VRF

```

<TabsContent sharedStore="vrfMethod" client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">

```

VRF gas costs are calculated based on the amount of verification gas and callback gas used, multiplied by the gas price:

$$(\text{Gas price} \times (\text{Verification gas} + \text{Callback gas})) = \text{total gas cost}$$

For VRF Arbitrum transactions, add a buffer to estimate the additional L1 cost:

```

(L2GasPrice
  (Verification gas
    + Callback gas
    + L1 calldata gas buffer))) = total estimated gas cost

```

To calculate the L1 callback gas buffer:

- calldataSizeBytes: A static size for the transaction's calldata in bytes.
Total: 720 bytes.
 - The amount Arbitrum adds to account for the static part of the transaction, fixed at 140 bytes.
 - The size of an ABI-encoded VRF V2 fulfillment, fixed at 580 bytes (fulfillmentTxSizeBytes).
- L1PricePerByte: The estimated cost of posting 1 byte of data on L1, which varies with L1 network gas prices.
- L2GasPrice: The L2 gas price, which varies with L2 network gas prices.

$$\begin{aligned} \text{L1 calldata gas buffer} &= (\text{calldataSizeBytes} \times \text{L1PricePerByte}) / \text{L2GasPrice} \\ &= (720 \times \text{L1PricePerByte}) / \text{L2GasPrice} \end{aligned}$$

This conversion allows us to estimate the L1 callback gas cost and incorporate it into the overall L2 gas estimate. You can add this estimated L1 callback gas buffer directly to the verification and callback gas:

```

(L2GasPrice
  (Verification gas
    + Callback gas
    + ((calldataSizeBytes \times L1PricePerByte) / L2GasPrice)))) = total estimated
gas cost

```

```

</Fragment>
<Fragment slot="panel.2">

```

VRF gas costs are calculated based on the amount of verification gas, callback gas and wrapper gas overhead, multiplied by the gas price:

$$(\text{Gas price} \times (\text{Verification gas} + \text{Callback gas} + \text{Wrapper gas overhead})) = \text{total gas cost}$$

For VRF Arbitrum transactions, add a buffer to estimate the additional L1 cost:

```
(L2GasPrice
  (Verification gas
    + Callback gas
    + Wrapper gas overhead
    + L1 calldata gas buffer))) = total estimated gas cost
```

To calculate the L1 callback gas buffer:

- calldataSizeBytes: A static size for the transaction's calldata in bytes. Total: 720 bytes.
- The amount Arbitrum adds to account for the static part of the transaction, fixed at 140 bytes.
- The size of an ABI-encoded VRF V2 fulfillment, fixed at 580 bytes (fulfillmentTxSizeBytes).
- L1PricePerByte: The estimated cost of posting 1 byte of data on L1, which varies with L1 network gas prices.
- L2GasPrice: The L2 gas price, which varies with L2 network gas prices.

```
L1 calldata gas buffer = (calldataSizeBytes  L1PricePerByte) / L2GasPrice

= (720  L1PricePerByte) / L2GasPrice
```

This conversion allows us to estimate the L1 callback gas cost and incorporate it into the overall L2 gas estimate. You can add this estimated L1 callback gas buffer directly to the verification, callback, and wrapper gas:

```
(L2GasPrice
  (Verification gas
    + Callback gas
    + Wrapper gas overhead
    + ((calldataSizeBytes  L1PricePerByte) / L2GasPrice)))) = total estimated
gas cost
```

</Fragment>
</TabsContent>

Arbitrum and VRF gas estimation code

This sample extends the original Arbitrum gas estimation script to estimate gas costs for VRF subscription and direct funding requests on Arbitrum.

The following snippet shows only the VRF variables and calculations that were added to the Arbitrum gas estimation script. To learn more about how Arbitrum gas is calculated, refer to the Arbitrum gas estimation tutorial. To run this code, use the full Arbitrum gas estimation script that includes VRF calculations.

```
typescript
// VRF variables and calculations
// -----
// Full script:
https://github.com/smartcontractkit/smart-contract-examples/tree/main/vrf-
arbitrum-gas-estimation
// -----
// Estimated upper bound of verification gas for VRF subscription.
// To see an estimate with an average amount of verification gas,
// adjust this to 115000.
const maxVerificationGas = 200000
```

```

// The L1 Calldata size includes:
// Arbitrum's static 140 bytes for transaction metadata
// VRF V2's static 580 bytes, the size of a fulfillment's calldata abi-encoded
in bytes
// (from sfulfillmentTxSizeBytes in VRFV2Wrapper.sol)
const VRFCallDataSizeBytes = 140 + 580

// For direct funding only. Coordinator gas is verification gas
const wrapperGasOverhead = 40000
const coordinatorGasOverhead = 90000

// VRF user settings
const callbackGasLimit = 175000

// Estimate VRF L1 buffer
const VRFL1CostEstimate = L1P.mul(VRFCallDataSizeBytes)
const VRFL1Buffer = VRFL1CostEstimate.div(P)

// VRF Subscription gas estimate
// L2 gas price (P) (maxVerificationGas + callbackGasLimit + VRFL1Buffer)
const VRFL2SubscriptionGasSubtotal = BigNumber.from(maxVerificationGas +
callbackGasLimit)
const VRFL2SubscriptionGasTotal = VRFL2SubscriptionGasSubtotal.add(VRFL1Buffer)
const VRFL2SubscriptionGasEstimate = P.mul(VRFL2SubscriptionGasTotal)

// VRF Direct funding gas estimate
// L2 gas price (P) (coordinatorGasOverhead + callbackGasLimit +
wrapperGasOverhead + VRFL1Buffer)
const VRFL2DirectFundingGasSubtotal = BigNumber.from(coordinatorGasOverhead +
wrapperGasOverhead + callbackGasLimit)
const VRFL2DirectFundingGasTotal = VRFL2DirectFundingGasSubtotal.add(VRFL1Buffer)
const VRFL2DirectFundingGasEstimate = P.mul(VRFL2DirectFundingGasTotal)

# getting-started.mdx:

---
section: legacy
date: Last Modified
title: "Getting Started with Chainlink VRF"
excerpt: "Using Chainlink VRF"
whatsnext:
{
  "Get a Random Number": "/vrf/v2/subscription/examples/get-a-random-number",
  "Programmatic Subscription": "/vrf/v2/subscription/examples/programmatic-
subscription",
  "Security Considerations": "/vrf/v2/security",
  "Best Practices": "/vrf/v2/best-practices",
  "Supported Networks": "/vrf/v2/subscription/supported-networks",
}
metadata:
  title: "Getting Started with Chainlink VRF V2"
  description: "Learn how to use randomness in your smart contracts using
Chainlink VRF."
  image: "/files/2a242f1-link.png"
---

import { Aside, CodeSample } from "@components"
import { YouTube } from "@astro-community/astro-embed-youtube"

<Aside type="note" title="Newer version of VRF available">
  Try the Getting Started with Chainlink VRF V2.5 tutorial.

```

</Aside>

<YouTube id="https://www.youtube.com/watch?v=AcDRNEKY1L0" />

<p style="text-align:center">VRF v2 - Developer Walkthrough</p>

In this guide, you will learn about generating randomness on blockchains. This includes learning how to implement a Request and Receive cycle with Chainlink oracles and how to consume random numbers with Chainlink VRF in smart contracts.

How is randomness generated on blockchains? What is Chainlink VRF?

Randomness is very difficult to generate on blockchains. This is because every node on the blockchain must come to the same conclusion and form a consensus. Even though random numbers are versatile and useful in a variety of blockchain applications, they cannot be generated natively in smart contracts. The solution to this issue is Chainlink VRF, also known as Chainlink Verifiable Random Function.

What is the Request and Receive cycle?

The Data Feeds Getting Started guide explains how to consume Chainlink Data Feeds, which consist of reference data posted onchain by oracles. This data is stored in a contract and can be referenced by consumers until the oracle updates the data again.

Randomness, on the other hand, cannot be reference data. If the result of randomness is stored onchain, any actor could retrieve the value and predict the outcome. Instead, randomness must be requested from an oracle, which generates a number and a cryptographic proof. Then, the oracle returns that result to the contract that requested it. This sequence is known as the Request and Receive cycle.

What is the payment process for generating a random number?

VRF requests receive funding from subscription accounts. The Subscription Manager lets you create an account and pre-pay for VRF requests, so that funding of all your application requests are managed in a single location. To learn more about VRF requests funding, see Subscriptions limits.

How can I use Chainlink VRF?

To see a basic implementation of Chainlink VRF, see Get a Random Number. In this section, you will create an application that uses Chainlink VRF to generate randomness. The contract used in this application has a Game of Thrones theme.

After the contract requests randomness from Chainlink VRF, the result of the randomness will transform into a number between 1 and 20, mimicking the rolling of a 20 sided die. Each number represents a Game of Thrones house. If the dice land on the value 1, the user is assigned house Targaryan, 2 for Lannister, and so on. A full list of houses can be found here.

When rolling the dice, it uses an address variable to track which address is assigned to each house.

The contract has the following functions:

- rollDice: This submits a randomness request to Chainlink VRF
- fulfillRandomWords: The function that the Oracle uses to send the result back
- house: To see the assigned house of an address

Note: to jump straight to the entire implementation, you can open the VRFD20.sol contract in remix.

Create and fund a subscription

Chainlink VRF requests receive funding from subscription accounts. The Subscription Manager lets you create an account and pre-pay your use of Chainlink VRF requests. For this example, create a new subscription on the Sepolia testnet as explained [here](#).

Importing VRFConsumerBaseV2 and VRFCoordinatorV2Interface

Chainlink maintains a library of contracts that make consuming data from oracles easier. For Chainlink VRF, you will use:

- VRFConsumerBaseV2 that must be imported and extended from the contract that you create.
- VRFCoordinatorV2Interface that must be imported to communicate with the VRF coordinator.

```
{/ prettier-ignore /}  
solidity  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.7;  
  
import  
"@chainlink/contracts/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol";  
import "@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";  
  
contract VRFD20 is VRFConsumerBaseV2 {  
  
}
```

Contract variables

This example is adapted for Sepolia testnet but you can change the configuration and make it run for any supported network.

```
{/ prettier-ignore /}  
solidity  
uint64 ssubscriptionId;  
address sowner;  
VRFCoordinatorV2Interface COORDINATOR;  
address vrfCoordinator = 0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625;  
bytes32 skeyHash =  
0x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c;  
uint32 callbackGasLimit = 40000;  
uint16 requestConfirmations = 3;  
uint32 numWords = 1;
```

- uint64 ssubscriptionId: The subscription ID that this contract uses for funding requests. Initialized in the constructor.
- address sowner: The address of the owner of the contract that you will deploy. This is initialized in the constructor, and it will be the address you use when deploying the contract.
- VRFCoordinatorV2Interface COORDINATOR: The address of the Chainlink VRF Coordinator contract that this contract will use. Initialized in the constructor.
- address vrfCoordinator: The address of the Chainlink VRF Coordinator contract.
- bytes32 skeyHash: The gas lane key hash value, which is the maximum gas price you are willing to pay for a request in wei. It functions as an ID of the offchain VRF job that runs in response to requests.
- uint32 callbackGasLimit: The limit for how much gas to use for the callback request to your contract's fulfillRandomWords function. It must be less than the

maxGasLimit on the coordinator contract. Adjust this value for larger requests depending on how your fulfillRandomWords function processes and stores the received random values. If your callbackGasLimit is not sufficient, the callback will fail and your subscription is still charged for the work done to generate your requested random values.

- uint16 requestConfirmations: How many confirmations the Chainlink node should wait before responding. The longer the node waits, the more secure the random value is. It must be greater than the minimumRequestBlockConfirmations limit on the coordinator contract.

- uint32 numWords: How many random values to request. If you can use several random values in a single callback, you can reduce the amount of gas that you spend per random value. In this example, each transaction requests one random value.

To keep track of addresses that roll the dice, the contract uses mappings. Mappings are unique key-value pair data structures similar to hash tables in Java.

```
{/ prettier-ignore /}  
solidity  
mapping(uint256 => address) private srollers;  
mapping(address => uint256) private sresults;
```

- srollers stores a mapping between the requestId (returned when a request is made), and the address of the roller. This is so the contract can keep track of who to assign the result to when it comes back.
- sresults stores the roller and the result of the dice roll.

Initializing the contract

The coordinator and subscription id must be initialized in the constructor of the contract. To use VRFConsumerBaseV2 properly, you must also pass the VRF coordinator address into its constructor.

The address that creates the smart contract is the owner of the contract. the modifier onlyOwner() checks that only the owner is allowed to do some tasks.

```
{/ prettier-ignore /}  
solidity  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.7;  
  
import  
"@chainlink/contracts/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol";  
import "@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";  
  
contract VRFD20 is VRFConsumerBaseV2 {  
    // variables  
    // ...  
  
    // constructor  
    constructor(uint64 subscriptionId) VRFConsumerBaseV2(vrfCoordinator) {  
        COORDINATOR = VRFCoordinatorV2Interface(vrfCoordinator);  
        sowner = msg.sender;  
        ssubscriptionId = subscriptionId;  
    }  
  
    //...  
    modifier onlyOwner() {  
        require(msg.sender == sowner);  
        ;  
    }  
}
```

rollDice function

The rollDice function will complete the following tasks:

1. Check if the roller has already rolled since each roller can only ever be assigned to a single house.
1. Request randomness by calling the VRF coordinator.
1. Store the requestId and roller address.
1. Emit an event to signal that the dice is rolling.

You must add a ROLLINPROGRESS constant to signify that the dice has been rolled but the result is not yet returned. Also add a DiceRolled event to the contract.

Only the owner of the contract can execute the rollDice function.

```

{
  // prettier-ignore
  solidity
  // SPDX-License-Identifier: MIT
  pragma solidity ^0.8.7;

  import
  "@chainlink/contracts/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol";
  import "@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";

  contract VRFD20 is VRFConsumerBaseV2 {
    // variables
    uint256 private constant ROLLINPROGRESS = 42;
    // ...

    // events
    event DiceRolled(uint256 indexed requestId, address indexed roller);
    // ...

    // ...
    // { constructor }
    // ...

    // rollDice function
    function rollDice(address roller) public onlyOwner returns (uint256
requestId) {
      require(sresults[roller] == 0, "Already rolled");
      // Will revert if subscription is not set and funded.
      requestId = COORDINATOR.requestRandomWords(
        skeyHash,
        ssubscriptionId,
        requestConfirmations,
        callbackGasLimit,
        numWords
      );

      srollers[requestId] = roller;
      sresults[roller] = ROLLINPROGRESS;
      emit DiceRolled(requestId, roller);
    }
  }
}

```

fulfillRandomWords function

fulfillRandomWords is a special function defined within the VRFConsumerBaseV2 contract that our contract extends from. The coordinator sends the result of our generated randomWords back to fulfillRandomWords. You will implement some functionality here to deal with the result:

1. Change the result to a number between 1 and 20 inclusively. Note that randomWords is an array that could contain several random values. In this example, request 1 random value.
1. Assign the transformed value to the address in the sresults mapping variable.
1. Emit a DiceLanded event.

```

{/ prettier-ignore /}
solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

import
"@chainlink/contracts/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol";
import "@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";

contract VRFD20 is VRFConsumerBaseV2 {
    // ...
    // { variables }
    // ...

    // events
    // ...
    event DiceLanded(uint256 indexed requestId, uint256 indexed result);

    // ...
    // { constructor }
    // ...

    // ...
    // { rollDice function }
    // ...

    // fulfillRandomWords function
    function fulfillRandomWords(uint256 requestId, uint256[] memory randomWords)
internal override {

        // transform the result to a number between 1 and 20 inclusively
        uint256 d20Value = (randomWords[0] % 20) + 1;

        // assign the transformed value to the address in the sresults mapping
variable
        sresults[srollers[requestId]] = d20Value;

        // emitting event to signal that dice landed
        emit DiceLanded(requestId, d20Value);
    }
}

```

house function

Finally, the house function returns the house of an address.

To have a list of the house's names, create the getHouseName function that is called in the house function.

```

{/ prettier-ignore /}
solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

import
"@chainlink/contracts/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol";

```

```

import "@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";

contract VRFD20 is VRFConsumerBaseV2 {
    // ...
    // { variables }
    // ...

    // ...
    // { events }
    // ...

    // ...
    // { constructor }
    // ...

    // ...
    // { rollDice function }
    // ...

    // ...
    // { fulfillRandomWords function }
    // ...

    // house function
    function house(address player) public view returns (string memory) {
        // dice has not yet been rolled to this address
        require(sresults[player] != 0, "Dice not rolled");

        // not waiting for the result of a thrown dice
        require(sresults[player] != ROLLINPROGRESS, "Roll in progress");

        // returns the house name from the name list function
        return getHouseName(sresults[player]);
    }

    // getHouseName function
    function getHouseName(uint256 id) private pure returns (string memory) {
        // array storing the list of house's names
        string[20] memory houseNames = [
            "Targaryen",
            "Lannister",
            "Stark",
            "Tyrell",
            "Baratheon",
            "Martell",
            "Tully",
            "Bolton",
            "Greyjoy",
            "Arryn",
            "Frey",
            "Mormont",
            "Tarley",
            "Dayne",
            "Umber",
            "Valeryon",
            "Manderly",
            "Clegane",
            "Glover",
            "Karstark"
        ];

        // returns the house name given an index
        return houseNames[id - 1];
    }
}

```

```
}
```

```
{/ prettier-ignore /}  
<CodeSample src="samples/VRF/VRFD20.sol" showButtonOnly/>
```

You have now completed all necessary functions to generate randomness and assign the user a Game of Thrones house. We've added a few helper functions in there to make using the contract easier and more flexible. You can deploy and interact with the complete contract in Remix.

How do I deploy to testnet?

You will deploy this contract on the Sepolia test network. You must have some Sepolia testnet ETH in your MetaMask account to pay for the gas. Testnet ETH is also available from several public faucets.

This deployment is slightly different than the example in the Deploy Your First Contract guide. In this case, you pass in parameters to the constructor upon deployment.

Once compiled, you'll see a dropdown menu that looks like this in the deploy pane:

```
!Remix contract selected
```

Select the VRFD20 contract or the name that you gave to your contract.

Click the caret arrow on the right hand side of Deploy to expand the parameter fields, and paste your subscription ID.

```
!Remix contract parameters to deploy
```

Then click the Deploy button and use your MetaMask account to confirm the transaction.

```
<Aside type="note" title="Address, Key Hashes and more">
```

```
  For a full reference of the addresses, key hashes and fees for each network,  
  see VRF Supported  
  Networks.  
</Aside>
```

At this point, your contract should be successfully deployed. However, it can't request anything because it is not yet approved to use the LINK balance in your subscription. If you click rollDice, the transaction will revert.

How do I add my contract to my subscription account?

After you deploy your contract, you must add it as an approved consumer contract so it can use the subscription balance when requesting for randomness. Go to the Subscription Manager and add your deployed contract address to the list of consumers. Find your contract address in Remix under Deployed Contracts on the bottom left.

```
!Remix contract address
```

How do I test rollDice?

After you open the deployed contract tab in the bottom left, the function buttons are available. Find rollDice and click the caret to expand the parameter fields. Enter an Ethereum address to specify a "dice roller", and click 'rollDice'.

It takes a few minutes for the transaction to confirm and the response to be

sent back. You can get your house by clicking the house function button with the address passed in rollDice. After the response is sent back, you'll be assigned a Game of Thrones house!

Further Reading

To read more about generating random numbers in Solidity, read our blog posts:

- 35+ Blockchain RNG Use Cases Enabled by Chainlink VRF
- How to Build Dynamic NFTs on Polygon
- Chainlink VRF v2 Now Live on Ethereum Mainnet

```
# security.mdx:
```

```
---
section: legacy
date: Last Modified
title: "VRF Security Considerations"
---
```

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
```

Gaining access to high quality randomness onchain requires a solution like Chainlink's VRF, but it also requires you to understand some of the ways that miners or validators can potentially manipulate randomness generation. Here are some of the top security considerations you should review in your project.

- Use requestId to match randomness requests with their fulfillment in order
- Choose a safe block confirmation time, which will vary between blockchains
- Do not allow re-requesting or cancellation of randomness
- Don't accept bids/bets/inputs after you have made a randomness request
- The fulfillRandomWords function must not revert
- Use VRFConsumerBaseV2 in your contract to interact with the VRF service

Use requestId to match randomness requests with their fulfillment in order

If your contract could have multiple VRF requests in flight simultaneously, you must ensure that the order in which the VRF fulfillments arrive cannot be used to manipulate your contract's user-significant behavior.

Blockchain miners/validators can control the order in which your requests appear onchain, and hence the order in which your contract responds to them.

For example, if you made randomness requests A, B, C in short succession, there is no guarantee that the associated randomness fulfillments will also be in order A, B, C. The randomness fulfillments might just as well arrive at your contract in order C, A, B or any other order.

We recommend using the requestId to match randomness requests with their corresponding fulfillments.

Choose a safe block confirmation time, which will vary between blockchains

In principle, miners/validators of your underlying blockchain could rewrite the chain's history to put a randomness request from your contract into a different block, which would result in a different VRF output. Note that this does not enable a miner to determine the random value in advance. It only enables them to get a fresh random value that might or might not be to their advantage. By way of analogy, they can only re-roll the dice, not predetermine or predict which side it will land on.

You must choose an appropriate confirmation time for the randomness requests you make. Confirmation time is how many blocks the VRF service waits before writing

a fulfillment to the chain to make potential rewrite attacks unprofitable in the context of your application and its value-at-risk.

Do not allow re-requesting or cancellation of randomness

Any re-request or cancellation of randomness is an incorrect use of VRFv2. dApps that implement the ability to cancel or re-request randomness for specific commitments must consider the additional attack vectors created by this capability. For example, you must prevent the ability for any party to discard unfavorable randomness.

Don't accept bids/bets/inputs after you have made a randomness request

Consider the example of a contract that mints a random NFT in response to a user's actions.

The contract should:

1. Record whatever actions of the user may affect the generated NFT.
1. Stop accepting further user actions that might affect the generated NFT and issue a randomness request.
1. On randomness fulfillment, mint the NFT.

Generally speaking, whenever an outcome in your contract depends on some user-supplied inputs and randomness, the contract should not accept any additional user-supplied inputs after it submits the randomness request.

Otherwise, the cryptoeconomic security properties may be violated by an attacker that can rewrite the chain.

fulfillRandomWords must not revert

If your fulfillRandomWords() implementation reverts, the VRF service will not attempt to call it a second time. Make sure your contract logic does not revert. Consider simply storing the randomness and taking more complex follow-on actions in separate contract calls made by you, your users, or an Automation Node.

Use VRFConsumerBaseV2 in your contract, to interact with the VRF service

If you implement the subscription method, use VRFConsumerBaseV2. It includes a check to ensure the randomness is fulfilled by VRFCoordinatorV2. For this reason, it is a best practice to inherit from VRFConsumerBaseV2. Similarly, don't override rawFulfillRandomness.

Use VRFv2WrapperConsumer.sol in your contract, to interact with the VRF service

If you implement the direct funding method, use VRFv2WrapperConsumer. It includes a check to ensure the randomness is fulfilled by the VRFV2Wrapper. For this reason, it is a best practice to inherit from VRFv2WrapperConsumer. Similarly, don't override rawFulfillRandomWords.

```
# index.mdx:
```

```
---
```

```
section: vrf
```

```
date: Last Modified
```

```
title: "Direct Funding Method"
```

```
isIndex: true
```

```
whatsnext:
```

```
{
```

```
  "Get a Random Number": "/vrf/v2/direct-funding/examples/get-a-random-number",
```

```
  "Supported Networks": "/vrf/v2/direct-funding/supported-networks",
```



```

    }
  metadata:
    title: "Generate Random Numbers for Smart Contracts using Chainlink VRF v2 - Direct funding method"
    description: "Learn how to securely generate random numbers for your smart contract with Chainlink VRF v2. This guide uses the Direct funding method."
    ---

import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
import { Aside, ClickToZoom } from "@components"

<Aside type="tip" title="VRF V2.5 Direct Funding Method">
  Refer to the VRF V2.5 Direct Funding Method Introduction page to learn how the direct funding method works in VRF V2.5. To compare V2.5 and V2, refer to the migration guide.
</Aside>

<VrfCommon callout="directFunding" />

```

This guide explains how to generate random numbers using the direct funding method. This method doesn't require a subscription and is optimal for one-off requests for randomness. This method also works best for applications where your end-users must pay the fees for VRF because the cost of the request is determined at request time.

VRF direct funding

Unlike the subscription method, the direct funding method does not require you to create subscriptions and pre-fund them. Instead, you must directly fund consuming contracts with LINK tokens before they request randomness. Because the consuming contract directly pays the LINK for the request, the cost is calculated during the request and not during the callback when the randomness is fulfilled. Learn how to estimate costs.

Request and receive data

Requests to Chainlink VRF v2 follow the request and receive data cycle. This end-to-end diagram shows each step in the lifecycle of a VRF direct funding request:

```
<ClickToZoom src="/images/vrf/v2-direct-funding-e2e.webp" />
```

Two types of accounts exist in the Ethereum ecosystem, and both are used in VRF:

- EOA (Externally Owned Account): An externally owned account that has a private key and can control a smart contract. Transactions can be initiated only by EOAs.
- Smart contract: A smart contract that does not have a private key and executes what it has been designed for as a decentralized application.

The Chainlink VRF v2 solution uses both offchain and onchain components:

- VRF v2 Wrapper (onchain component): A wrapper for the VRF Coordinator that provides an interface for consuming contracts.
- VRF v2 Coordinator (onchain component): A contract designed to interact with the VRF service. It emits an event when a request for randomness is made, and then verifies the random number and proof of how it was generated by the VRF service.
- VRF service (offchain component): Listens for requests by subscribing to the VRF Coordinator event logs and calculates a random number based on the block hash and nonce. The VRF service then sends a transaction to the VRFCoordinator including the random number and a proof of how it was generated.

Set up your contract and request

Set up your consuming contract:

1. Your contract must inherit VRFV2WrapperConsumerBase.

1. Your contract must implement the fulfillRandomWords function, which is the callback VRF function. Here, you add logic to handle the random values after they are returned to your contract.

1. Submit your VRF request by calling the requestRandomness function in the VRFV2WrapperConsumerBase contract. Include the following parameters in your request:

- requestConfirmations: The number of block confirmations the VRF service will wait to respond. The minimum and maximum confirmations for your network can be found [here](#).
- callbackGasLimit: The maximum amount of gas to pay for completing the callback VRF function.
- numWords: The number of random numbers to request. You can find the maximum number of random values per request for your network in the [Supported networks](#) page.

How VRF processes your request

After you submit your request, it is processed using the Request & Receive Data cycle:

1. The consuming contract calls the VRFV2Wrapper calculateRequestPrice function to estimate the total transaction cost to fulfill randomness. Learn how to estimate transaction costs.

1. The consuming contract calls the LinkToken transferAndCall function to pay the wrapper with the calculated request price. This method sends LINK tokens and executes the VRFV2Wrapper onTokenTransfer logic.

1. The VRFV2Wrapper's onTokenTransfer logic triggers the VRF Coordinator requestRandomWords function to request randomness.

1. The VRF coordinator emits an event.

1. The VRF service picks up the event and waits for the specified number of block confirmations to respond back to the VRF coordinator with the random values and a proof (requestConfirmations).

1. The VRF coordinator verifies the proof onchain, then it calls back the wrapper contract's fulfillRandomWords function.

1. Finally, the VRF Wrapper calls back your consuming contract.

Limits

You can see the configuration for each network on the [Supported networks](#) page. You can also view the full configuration for each VRF v2 Wrapper contract directly in Etherscan. As an example, view the [Ethereum Mainnet VRF v2 Wrapper](#) contract configuration by calling getConfig function.

- Each wrapper has a maxNumWords parameter that limits the maximum number of random values you can receive in each request.
- The maximum allowed callbackGasLimit value for your requests is defined in the Coordinator contract supported networks page. Because the VRF v2 Wrapper adds an overhead, your callbackGasLimit must not exceed maxGasLimit - wrapperGasOverhead. Learn more about estimating costs.

supported-networks.mdx:

```
section: legacy
date: Last Modified
title: "Supported Networks - Direct Funding Method"
metadata:
  title: "Chainlink VRF Contract Addresses"
  linkToWallet: true
  image: "/files/OpenGraphV3.png"
```

```
import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"
import ResourcesCallout from
"@features/resources/callouts/ResourcesCallout.astro"
import { Address, Aside, CopyText } from "@components"
```

```
<Vrf25Common callout="supportednetworks" />
```

Chainlink VRF allows you to integrate provably fair and verifiably random data in your smart contract.

For implementation details, read [Introduction to Chainlink VRF v2 Direct funding method](#).

Wrapper parameters

These parameters are configured in the VRF v2 Wrapper contract. You can view these values by running `getConfig` on the VRF v2 Wrapper or by viewing the VRF v2 Wrapper contract in a blockchain explorer.

- `uint32 stalenessSeconds`: How long the VRF v2 Wrapper waits until we consider the ETH/LINK price used for converting gas costs to LINK is stale and use `fallbackWeiPerUnitLink`.
- `uint32 wrapperGasOverhead`: The gas overhead of the VRF v2 Wrapper's `fulfillRandomWords` function.
- `uint32 coordinatorGasOverhead`: The gas overhead of the coordinator's `fulfillRandomWords` function.
- `uint8 maxNumWords`: Maximum number of words that can be requested in a single wrapped VRF request.

Coordinator parameters

Some parameters are important to know and are configured in the coordinator contract. You can view these values by running `getConfig` on the coordinator or by viewing the coordinator contract in a blockchain explorer.

- `uint16 minimumRequestConfirmations`: The minimum number of confirmation blocks on VRF requests before oracles respond
- `uint32 maxGasLimit`: The maximum gas limit supported for a `fulfillRandomWords` callback. Note that you still need to subtract the `wrapperGasOverhead` for the accurate limit, as explained in [Direct funding limits](#).

Fee parameters

Fee parameters are configured in the VRF v2 Wrapper and the VRF v2 Coordinator contracts and specify the premium you pay per request in addition to the gas cost for the transaction. You can view them by running `getConfig` on the VRF v2 Wrapper:

- The `uint32 fulfillmentFlatFeeLinkPPM` parameter is a flat fee and defines the fees per request specified in millionths of LINK.
- The `uint8 wrapperPremiumPercentage` parameter defines the premium ratio in

percentage. For example, a value of 0 indicates no premium. A value of 15 indicates a 15% premium.

The details for calculating the total transaction cost can be found [here](#).

Configurations

VRF v2 coordinators for direct funding are available on several networks. To see a list of coordinators for subscription funding, see the [Subscription Configurations](#) page.

<ResourcesCallout callout="bridgeRisks" />

Ethereum mainnet

<Vrf25Common callout="supportednetworks" />

Item	Value

LINK Token	<Address contractUrl="https://etherscan.io/token/0x514910771AF9Ca656af840dff83E8264EcF986CA" urlId="10x514910771AF9Ca656af840dff83E8264EcF986CA" urlClass="erc-token-address"/>
VRF Wrapper	<Address contractUrl="https://etherscan.io/address/0x5A861794B927983406fCE1D062e00b9368d97Df6" />
VRF Coordinator	<Address contractUrl="https://etherscan.io/address/0x271682DEB8C4E0901D1a1550aD2e64D568E69909" />
Wrapper Premium Percentage	0
Coordinator Flat Fee	0.25 LINK
Minimum Confirmations	3
Maximum Confirmations	200
Maximum Random Values	10
Wrapper Gas overhead	40000
Coordinator Gas Overhead	90000

Sepolia testnet

<Vrf25Common callout="supportednetworks" />

Testnet LINK and ETH are available from [faucets.chain.link](https://faucets.chain.link/sepolia).
Testnet ETH is also available from several public [ETH faucets](https://faucetlink.to/sepolia).

Item	Value


```

----- |
| LINK Token | <Address
contractUrl="https://sepolia.etherscan.io/token/0x779877A7B0D9E8603169DdbD7836e4
78b4624789" urlId="111551110x779877A7B0D9E8603169DdbD7836e478b4624789"
urlClass="erc-token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://sepolia.etherscan.io/address/0xab18414CD93297B0d12ac29E63Ca
20f515b3DB46" />
| VRF Coordinator | <Address
contractUrl="https://sepolia.etherscan.io/address/0x8103B0A8A00be2DDC778e6e7eaa2
1791Cd364625" />
| Wrapper Premium Percentage | 0
| Coordinator Flat Fee | 0.25 LINK
| Minimum Confirmations | 3
| Maximum Confirmations | 200
| Maximum Random Values | 10
| Wrapper Gas overhead | 40000
| Coordinator Gas Overhead | 90000

```

BNB Chain mainnet

```
<Vrf25Common callout="supportednetworks" />
```

```
<Aside type="tip" title="Important">
```

The LINK provided by the BNB Chain Bridge is not ERC-677 compatible, so cannot be used with Chainlink oracles. However, it can be converted to the official LINK token on BNB Chain using Chainlink's PegSwap service.

```
</Aside>
```

```

| Item | Value
| ----- |

```

```

----- |
| LINK Token | <Address
contractUrl="https://bscscan.com/token/0x404460C6A5EdE2D891e8297795264fDe62ADBB7
5" urlId="560x404460C6A5EdE2D891e8297795264fDe62ADBB75" urlClass="erc-token-
address"/> |
| VRF Wrapper | <Address
contractUrl="https://bscscan.com/address/0x721DFbc5Cfe53d32ab00A9bdFa605d3b8E1f3
f42" />
| VRF Coordinator | <Address
contractUrl="https://bscscan.com/address/0xc587d9053cd1118f25F645F9E08BB98c9712A
4EE" />
| Wrapper Premium Percentage | 0
| Coordinator Flat Fee | 0.005 LINK
| Minimum Confirmations | 3

```

Maximum Confirmations	200
Maximum Random Values	10
Wrapper Gas overhead	40000
Coordinator Gas Overhead	90000

BNB Chain testnet

<Vrf25Common callout="supportednetworks" />

Testnet LINK is available from <https://faucets.chain.link/bnb-chain-testnet>

Item	Value

LINK Token	<Address contractUrl="https://testnet.bscscan.com/address/0x84b9B910527Ad5C03A9Ca831909E21e236EA7b06" urlId="970x84b9B910527Ad5C03A9Ca831909E21e236EA7b06" urlClass="erc-token-address"/>
VRF Wrapper	<Address contractUrl="https://testnet.bscscan.com/address/0x699d428ee890d55D56d5FC6e26290f3247A762bd" />
VRF Coordinator	<Address contractUrl="https://testnet.bscscan.com/address/0x6A2AAd07396B36Fe02a22b33cf443582f682c82f" />
Wrapper Premium Percentage	0
Coordinator Flat Fee	0.005 LINK
Minimum Confirmations	3
Maximum Confirmations	200
Maximum Random Values	10
Wrapper Gas overhead	40000
Coordinator Gas Overhead	90000

Polygon mainnet

<Vrf25Common callout="supportednetworks" />

<Aside type="tip" title="Important">

The LINK provided by the Polygon Bridge is not ERC-677 compatible, so cannot be used with Chainlink oracles. However, it can be converted to the official LINK token on Polygon using Chainlink's PegSwap service

</Aside>

Item	Value


```

----- |
| LINK Token | <Address
contractUrl="https://polygonscan.com/address/0xb0897686c545045aFc77CF20eC7A532E3
120E0F1" urlId="1370xb0897686c545045aFc77CF20eC7A532E3120E0F1" urlClass="erc-
token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://polygonscan.com/address/0x4e42f0adEB69203ef7Aa4B7c414e5b13
31c14dc" />
|
| VRF Coordinator | <Address
contractUrl="https://polygonscan.com/address/0xAE975071Be8F8eE67addBC1A82488F1C2
4858067" />
|
| Wrapper Premium Percentage | 0
|
| Coordinator Flat Fee | 0.0005 LINK
|
| Minimum Confirmations | 3
|
| Maximum Confirmations | 200
|
| Maximum Random Values | 10
|
| Wrapper Gas overhead | 40000
|
| Coordinator Gas Overhead | 90000
|

```

Polygon Amoy testnet

<Vrf25Common callout="supportednetworks" />

```

| Item | Value
| ----- |
| ----- |
| ----- |
|
| LINK Token | <Address
contractUrl="https://amoy.polygonscan.com/address/0x0fd9e8d3af1aaee056eb9e802c3a
762a667b1904" urlId="800020x0fd9e8d3af1aaee056eb9e802c3a762a667b1904"
urlClass="erc-token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://amoy.polygonscan.com/address/0x7E10652Cb79Ba97bC1D0F38a1e8F
aD8464a8a908" />
|
| 500 gwei Key Hash | <CopyText
text="0x3f631d5ec60a0ce16203bcd6aff7ffbc423e22e452786288e172d467354304c8"
code />
|
| Premium | 0.0005 LINK
|
| Max Gas Limit | 2,500,000
|
| Minimum Confirmations | 3
|
| Maximum Confirmations | 200
|
| Maximum Random Values | 10
|
| Wrapper Gas overhead | 40000
|
| Coordinator Gas Overhead | 90000
|

```

Avalanche mainnet

<Vrf25Common callout="supportednetworks" />

Item	Value

LINK Token	<Address contractUrl="https://snowtrace.io/address/0x5947BB275c521040051D82396192181b413227A3" urlId="431140x5947BB275c521040051D82396192181b413227A3" urlClass="erc-token-address"/>
VRF Wrapper	<Address contractUrl="https://snowtrace.io/address/0x721DFbc5Cfe53d32ab00A9bdFa605d3b8E1f3f42" />
VRF Coordinator	<Address contractUrl="https://snowtrace.io/address/0xd5D517aBE5cF79B7e95eC98dB0f027778aF634" />
Wrapper Premium Percentage	0
Coordinator Flat Fee	0.005 LINK
Minimum Confirmations	1
Maximum Confirmations	200
Maximum Random Values	10
Wrapper Gas overhead	40000
Coordinator Gas Overhead	90000

Avalanche Fuji testnet

<Vrf25Common callout="supportednetworks" />

Testnet LINK is available from <https://faucets.chain.link/fuji>

Item	Value

LINK Token	<Address contractUrl="https://testnet.snowtrace.io/address/0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" urlId="431130x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846" urlClass="erc-token-address"/>
VRF Wrapper	<Address contractUrl="https://testnet.snowtrace.io/address/0x9345AC54dA4D0B5Cda8CB749d8ef37e5F02BBb21" />
VRF Coordinator	<Address contractUrl="https://testnet.snowtrace.io/address/0x2eD832Ba664535e5886b75D64C46EB9a228C2610" />
Wrapper Premium Percentage	0

Coordinator Flat Fee	0.005 LINK
Minimum Confirmations	1
Maximum Confirmations	200
Maximum Random Values	10
Wrapper Gas overhead	40000
Coordinator Gas Overhead	90000

Fantom mainnet

<Aside type="tip" title="Important">

You must use ERC-677 LINK on Fantom. ERC-20 LINK will not work with Chainlink services. Fantom is not supported in VRF V2.5.

</Aside>

Item	Value

LINK Token	<Address contractUrl="https://ftmscan.com/token/0x6F43FF82CCA38001B6699a8AC47A2d0E66939407" urlId="2500x6F43FF82CCA38001B6699a8AC47A2d0E66939407" urlClass="erc-token-address"/>
VRF Wrapper	<Address contractUrl="https://ftmscan.com/address/0xeDA5B00fB33B13c730D004Cf5D1aDa1ac191Ddc2" />
VRF Coordinator	<Address contractUrl="https://ftmscan.com/address/0xd5D517aBE5cF79B7e95eC98dB0f0277788aFF634" />
Wrapper Premium Percentage	0
Coordinator Flat Fee	0.0005 LINK
Minimum Confirmations	1
Maximum Confirmations	200
Maximum Random Values	10
Wrapper Gas overhead	40000
Coordinator Gas Overhead	90000

Fantom testnet

<Aside type="note" title="Fantom Testnet Faucet">

Testnet LINK is available from <https://faucets.chain.link/fantom-testnet>.

</Aside>

Item	Value

```

| ----- |
|
| LINK Token | <Address
contractUrl="https://testnet.ftmscan.com/address/0xfaFedb041c0DD4fA2Dc0d87a6B097
9Ee6FA7af5F" urlId="40020xfaFedb041c0DD4fA2Dc0d87a6B0979Ee6FA7af5F"
urlClass="erc-token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://testnet.ftmscan.com/address/0x38336BDaE79747a1d2c4e6C67BBF3
82244287ca6" />
| VRF Coordinator | <Address
contractUrl="https://testnet.ftmscan.com/address/0xbd13f08b8352A3635218ab9418E34
0c60d6Eb418" />
| Wrapper Premium Percentage | 0
| Coordinator Flat Fee | 0.0005 LINK
| Minimum Confirmations | 1
| Maximum Confirmations | 200
| Maximum Random Values | 10
| Wrapper Gas overhead | 40000
| Coordinator Gas Overhead | 90000
|

```

Arbitrum mainnet

```
<Vrf25Common callout="supportednetworks" />
```

```

| Item | Value
| ----- |
|
| LINK Token | <Address
contractUrl="https://arbiscan.io/address/0xf97f4df75117a78c1A5a0DBb814Af92458539
FB4" urlId="421610xf97f4df75117a78c1A5a0DBb814Af92458539FB4" urlClass="erc-
token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://arbiscan.io/address/0x41034678D6C633D8a95c75e1138A360a28bA1
5d1" />
| VRF Wrapper | <Address
contractUrl="https://arbiscan.io/address/0x2D159AE3bFf04a10A355B608D22BDEC092e93
4fa" />
| Wrapper Premium Percentage | 0
| Coordinator Flat Fee | 0.0005 LINK
| Minimum Confirmations | 1
| Maximum Confirmations | 200
| Maximum Random Values | 10
|

```

Wrapper Gas overhead	40000
Coordinator Gas Overhead	90000

Arbitrum Sepolia testnet

<Vrf25Common callout="supportednetworks" />

Testnet LINK is available from <https://faucets.chain.link/arbitrum-sepolia>

Item	Value

LINK Token	<Address contractUrl="https://sepolia.arbiscan.io/address/0xb1D4538B4571d411F07960EF2838Ce337FE1E80E" urlId="4216140xb1D4538B4571d411F07960EF2838Ce337FE1E80E" urlClass="erc-token-address"/>
VRF Coordinator	<Address contractUrl="https://sepolia.arbiscan.io/address/0x50d47e4142598E3411aA864e08a44284e471AC6f" />
VRF Wrapper	<Address contractUrl="https://sepolia.arbiscan.io/address/0x1D3bb92db7659F2062438791F131CFA396dfb592" />
Wrapper Premium Percentage	0
Coordinator Flat Fee	0.0005 LINK
Minimum Confirmations	1
Maximum Confirmations	200
Maximum Random Values	10
Wrapper Gas overhead	40000
Coordinator Gas Overhead	90000

get-a-random-number.mdx:

```

---
section: legacy
date: Last Modified
title: "Get a Random Number with VRF V2"
whatsnext:
  {
    "Security Considerations": "/vrf/v2/security",
    "Best Practices": "/vrf/v2/best-practices",
    "Migrating from VRF v1": "/vrf/v2/direct-funding/migration-from-v1",
    "Supported Networks": "/vrf/v2/direct-funding/supported-networks",
  }
metadata:
  description: "How to generate a random number inside a smart contract using
Chainlink VRF v2 - Direct funding method."
---
```

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
import { Aside, CodeSample } from "@components"
```

```
<VrfCommon callout="directFunding" />
```

This guide explains how to get random values using a simple contract to request and receive random values from Chainlink VRF v2 without managing a subscription. To explore more applications of VRF, refer to our blog.

Requirements

This guide assumes that you know how to create and deploy smart contracts on Ethereum testnets using the following tools:

- The Remix IDE
- MetaMask
- Sepolia testnet ETH

If you are new to developing smart contracts on Ethereum, see the Getting Started guide to learn the basics.

Create and deploy a VRF v2 compatible contract

For this example, use the `VRFv2DirectFundingConsumer.sol` sample contract. This contract imports the following dependencies:

- `VRFV2WrapperConsumerBase.sol`([link](#))
- `ConfirmedOwner.sol`([link](#))

The contract also includes pre-configured values for the necessary request parameters such as `callbackGasLimit`, `requestConfirmations`, the number of random words `numWords`, the VRF v2 Wrapper address `wrapperAddress`, and the LINK token address `linkAddress`. You can change these parameters if you want to experiment on different testnets.

Build and deploy the contract on Sepolia.

1. Open the `VRFv2DirectFundingConsumer.sol` contract in Remix.

```
{/ prettier-ignore /}
<CodeSample src="samples/VRF/VRFv2DirectFundingConsumer.sol" showButtonOnly/>
```

1. On the Compile tab in Remix, compile the `VRFv2DirectFundingConsumer` contract.

1. Configure your deployment. On the Deploy tab in Remix, select the Injected Web3 Environment and select the `VRFv2DirectFundingConsumer` contract from the contract list.

1. Click the Deploy button to deploy your contract onchain. MetaMask opens and asks you to confirm the transaction.

1. After you deploy your contract, copy the address from the Deployed Contracts list in Remix. Before you can request randomness from VRF v2, you must fund your consuming contract with enough LINK tokens in order to request for randomness. Next, fund your contract.

Fund Your Contract

Requests for randomness will fail unless your consuming contract has enough LINK.

1. Acquire testnet LINK.

1. Fund your contract with testnet LINK. For this example, funding your contract

with 2 LINK should be sufficient.

Request random values

The deployed contract requests random values from Chainlink VRF, receives those values, builds a struct `RequestStatus` containing them, and stores the struct in a mapping `srequests`. Run the `requestRandomWords()` function on your contract to start the request.

1. Return to Remix and view your deployed contract functions in the Deployed Contracts list.

1. Click the `requestRandomWords()` function to send the request for random values to Chainlink VRF. MetaMask opens and asks you to confirm the transaction.

<Aside type="note" title="Set your gas limit in MetaMask">
Remix IDE doesn't set the right gas limit, so you must edit the gas limit in MetaMask within the Advanced gas controls settings.

For this example to work, set the gas limit to 400,000 in MetaMask.

First, enable Advanced gas controls in your MetaMask settings.

Before confirming your transaction in MetaMask, navigate to the screen where you can edit the gas limit: Select Site suggested > Advanced > Advanced gas controls and select Edit next to the Gas limit amount. Update the Gas limit value to 400000 and select Save. Finally, confirm the transaction.

</Aside>

After you approve the transaction, Chainlink VRF processes your request. Chainlink VRF fulfills the request and returns the random values to your contract in a callback to the `fulfillRandomWords()` function. At this point, a new key `requestId` is added to the mapping `srequests`. Depending on current testnet conditions, it might take a few minutes for the callback to return the requested random values to your contract.

1. To fetch the request ID of your request, call `lastRequestId()`.

1. After the oracle returns the random values to your contract, the mapping `srequests` is updated. The received random values are stored in `srequests[requestId].randomWords`.

1. Call `getRequestStatus()` and specify the `requestId` to display the random words.

<Aside type="note" title="Note on Requesting or Cancelling Randomness">
Do not allow re-requesting or cancellation of randomness. For more information, see the VRF Security Considerations page.
</Aside>

Analyzing the contract

In this example, the consuming contract uses static configuration parameters.

```
<CodeSample src="samples/VRF/VRFv2DirectFundingConsumer.sol" />
```

The parameters define how your requests will be processed. You can find the values for your network in the Supported networks page.

- `uint32 callbackGasLimit`: The limit for how much gas to use for the callback request to your contract's `fulfillRandomWords()` function. It must be less than the `maxGasLimit` limit on the coordinator contract minus the `wrapperGasOverhead`.

See the VRF v2 Direct funding limits for more details. Adjust this value for larger requests depending on how your fulfillRandomWords() function processes and stores the received random values. If your callbackGasLimit is not sufficient, the callback will fail and your consuming contract is still charged for the work done to generate your requested random values.

- uint16 requestConfirmations: How many confirmations the Chainlink node should wait before responding. The longer the node waits, the more secure the random value is. It must be greater than the minimumRequestBlockConfirmations limit on the coordinator contract.

- uint32 numWords: How many random values to request. If you can use several random values in a single callback, you can reduce the amount of gas that you spend per random value. The total cost of the callback request depends on how your fulfillRandomWords() function processes and stores the received random values, so adjust your callbackGasLimit accordingly.

The contract includes the following functions:

- requestRandomWords(): Takes your specified parameters and submits the request to the VRF v2 Wrapper contract.

- fulfillRandomWords(): Receives random values and stores them with your contract.

- getRequestStatus(): Retrieve request details for a given requestId.

- withdrawLink(): At any time, the owner of the contract can withdraw outstanding LINK balance from it.

<Aside type="note" title="Security Considerations">

Be sure to review your contracts to make sure they follow the best practices on the security considerations page.

</Aside>

Clean up

After you are done with this contract, you can retrieve the remaining testnet LINK to use with other examples.

1. Call withdrawLink() function. MetaMask opens and asks you to confirm the transaction. After you approve the transaction, the remaining LINK will be transferred from your consuming contract to your wallet address.

test-locally.mdx:

section: legacy

title: "Local testing using a Mock contract"

metadata:

description: "Example contract for generating random words using the VRF v2 direct funding method on your local blockchain using a mock contract."

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
import ContentCommon from "@features/common/ContentCommon.astro"
import { CodeSample, ClickToZoom, Aside } from "@components"
```

<VrfCommon callout="directFunding" />

This guide explains how to test Chainlink VRF v2 on a Remix IDE sandbox blockchain environment. Note: You can reuse the same logic on another

development environment, such as Hardhat or Truffle. For example, read the Hardhat Starter Kit `RandomNumberDirectFundingConsumer` unit tests.

<Aside type="caution" title="Test on public testnets thoroughly">

Even though local testing has several benefits, testing with a VRF mock covers the bare minimum of use cases. Make

sure to test your consumer contract thoroughly on public testnets.

</Aside>

Benefits of local testing

<ContentCommon section="localTestingBenefits" />

Testing logic

Complete the following tasks to test your VRF v2 consumer locally:

1. Deploy the `VRFCoordinatorV2Mock`. This contract is a mock of the `VRFCoordinatorV2` contract.
1. Deploy the `MockV3Aggregator` contract.
1. Deploy the `LinkToken` contract.
1. Deploy the `VRFV2Wrapper` contract.
1. Call the `VRFV2Wrapper` `setConfig` function to set wrapper specific parameters.
1. Fund the `VRFV2Wrapper` subscription.
1. Call the `VRFCoordinatorV2Mock` `addConsumer` function to add the wrapper contract to your subscription.
1. Deploy your VRF consumer contract.
1. Fund your consumer contract with LINK tokens.
1. Request random words from your consumer contract.
1. Call the `VRFCoordinatorV2Mock` `fulfillRandomWords` function to fulfill your consumer contract request.

Prerequisites

This guide will require you to finetune the gas limit when fulfilling requests. When writing, manually setting up the gas limits on RemixIDE is not supported, so you will use RemixIDE in conjunction with Metamask. Ganache lets you quickly fire up a personal Ethereum blockchain. If you still need to install Ganache, follow the official guide.

1. Once Ganache is installed, click the QUICKSTART button to start a local Ethereum node.

<ClickToZoom src="/images/vrf/mock/ganache.jpg" />

Note: Make sure to note the RPC server. In this example, the RPC server is `http://127.0.0.1:7545/`.

1. Follow the Metamask official guide to add a custom network manually.

<ClickToZoom src="/images/vrf/mock/metamaskcustomnetwork.jpg" />

1. Import a Ganache account into Metamask.

1. On Ganache, click on the key symbol of the first account:

<ClickToZoom src="/images/vrf/mock/ganacheshowkeys.jpg" />

1. Copy the private key:

<ClickToZoom src="/images/vrf/mock/ganacheprivatekey.jpg" />

1. Follow the Metamask official guide to import an account using a private key.

1. Your Metamask is connected to Ganache, and you should have access to the newly imported account.

<ClickToZoom src="/images/vrf/mock/metamaskganacheaccount.jpg" />{" "}

Testing

Open the contracts on RemixIDE

Open VRFCoordinatorV2Mock and compile in Remix:

<CodeSample src="samples/VRF/mock/VRFCoordinatorV2Mock.sol" />

Open MockV3Aggregator and compile in Remix:

<CodeSample src="samples/VRF/mock/MockV3Aggregator.sol" />

Open LinkToken and compile in Remix:

<CodeSample src="samples/VRF/mock/LinkToken.sol" />

Open VRFV2Wrapper and compile in Remix:

<CodeSample src="samples/VRF/mock/VRFV2Wrapper.sol" />

Open RandomNumberDirectFundingConsumerV2 and compile in Remix:

<CodeSample src="samples/VRF/mock/RandomNumberDirectFundingConsumerV2.sol" />

Your RemixIDE file explorer should display the opened contracts:

<ClickToZoom src="/images/vrf/mock/v2-df-remix-fileexplorer.jpg" />

Select the correct RemixIDE environment

Under DEPLOY & RUN TRANSACTIONS:

1. Set the Environment to Injected Provider - Metamask:

<ClickToZoom src="/images/vrf/mock/injectedmetamask.jpg" />

1. On Metamask, connect your Ganache account to the Remix IDE.

<ClickToZoom src="/images/vrf/mock/ganacheremixconnect.jpg" />

1. Click on Connect. The RemixIDE environment should be set to the correct environment, and the account should be the Ganache account.

<ClickToZoom src="/images/vrf/mock/remixenvinjected.jpg" />

Deploy VRFCoordinatorV2Mock

1. Open VRFCoordinatorV2Mock.sol.

1. Under DEPLOY & RUN TRANSACTIONS, select VRFCoordinatorV2Mock.

<ClickToZoom src="/images/vrf/mock/v2-deploymockdf.jpg" />

1. Under DEPLOY, fill in the BASEFEE and GASPRICELINK. These variables are used in the VRFCoordinatorV2Mock contract to represent the base fee and the gas price (in LINK tokens) for the VRF requests. You can set: BASEFEE=1000000000000000000 and GASPRICELINK=1000000000.

1. Click on transact to deploy the VRFCoordinatorV2Mock contract.

1. A Metamask popup will open. Click on Confirm.

1. Once deployed, you should see the VRFCoordinatorV2Mock contract under Deployed Contracts.

<ClickToZoom src="/images/vrf/mock/v2-deployedmockdf.jpg" />

1. Note the address of the deployed contract.

Deploy MockV3Aggregator

The MockV3Aggregator contract is designed for testing purposes, allowing you to simulate an oracle price feed without interacting with the existing Chainlink network.

1. Open MockV3Aggregator.sol.

1. Under DEPLOY & RUN TRANSACTIONS, select MockV3Aggregator.

<ClickToZoom src="/images/vrf/mock/v2-deployaggregatormock.jpg" />

1. Under DEPLOY, fill in DECIMALS and INITIALANSWER. These variables are used in the MockV3Aggregator contract to represent the number of decimals the aggregator's answer should have and the most recent price feed answer. You can set: DECIMALS=18 and INITIALANSWER=300000000000000000 (We are considering that 1 LINK = 0.003 native gas tokens).

1. Click on transact to deploy the MockV3Aggregator contract.

1. A Metamask popup will open. Click on Confirm.

1. Once deployed, you should see the MockV3Aggregator contract under Deployed Contracts.

<ClickToZoom src="/images/vrf/mock/v2-deployedaggregatormock.jpg" />

1. Note the address of the deployed contract.

Deploy LinkToken

The Chainlink VRF v2 direct funding method requires your consumer contract to pay for VRF requests in LINK. Therefore, you have to deploy the LinkToken contract to your local blockchain.

1. Open LinkToken.sol.

1. Under DEPLOY & RUN TRANSACTIONS, select LinkToken.

<ClickToZoom src="/images/vrf/mock/v2-deploylink.jpg" />

1. Under DEPLOY, click on transact to deploy the LinkToken contract.

1. A Metamask popup will open. Click on Confirm.

1. Once deployed, you should see the LinkToken contract under Deployed Contracts.

<ClickToZoom src="/images/vrf/mock/v2-deployedlink.jpg" />

1. Note the address of the deployed contract.

Deploy VRFV2Wrapper

As the VRF v2 direct funding end-to-end diagram explains, the VRFV2Wrapper acts as a wrapper for the coordinator contract.

1. Open VRFV2Wrapper.sol.

1. Under DEPLOY & RUN TRANSACTIONS, select VRFV2Wrapper.

<ClickToZoom src="/images/vrf/mock/v2-deploywrapper.jpg" />

1. Under DEPLOY, fill in LINK with the LinkToken contract address, LINKETHFEED with the MockV3Aggregator contract address, and COORDINATOR with the VRFCoordinatorV2Mock contract address.

1. click on transact to deploy the VRFV2Wrapper contract.

1. A Metamask popup will open. Click on Confirm.

1. Once deployed, you should see the VRFV2Wrapper contract under Deployed Contracts.

<ClickToZoom src="/images/vrf/mock/v2-deployedwrapper.jpg" />

1. Note the address of the deployed contract.

Configure the VRFV2Wrapper

1. Under Deployed Contracts, open the functions list of your deployed VRFV2Wrapper contract.

1. Click on setConfig and fill in wrapperGasOverhead with 60000, coordinatorGasOverhead with 52000, wrapperPremiumPercentage with 10, keyHash with 0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc, and maxNumWords with 10. Note on these variables:

1. wrapperGasOverhead: This variable reflects the gas overhead of the wrapper's fulfillRandomWords function. The cost for this gas is passed to the user.

1. coordinatorGasOverhead: This variable reflects the gas overhead of the coordinator's fulfillRandomWords function. The cost for this gas is billed to the VRFV2Wrapper subscription and must, therefore, be included in the VRF v2 direct funding requests pricing.

1. wrapperPremiumPercentage: This variable is the premium ratio in percentage. For example, a value of 0 indicates no premium. A value of 15 indicates a 15 percent premium.

1. keyHash: The gas lane key hash value is the maximum gas price you are willing to pay for a request in wei.

1. maxNumWords: This variable is the maximum number of words requested in a VRF v2 direct funding request.

<ClickToZoom src="/images/vrf/mock/v2-wrappersetconfig.jpg" />

1. click on transact.

1. A Metamask popup will open. Click on Confirm.

Fund the VRFV2Wrapper subscription

When deployed, the VRFV2Wrapper contract creates a new subscription and adds itself to the newly created subscription. If you started this guide from

scratch, the subscription ID should be 1.

1. Under Deployed Contracts, open the functions list of your deployed VRFCoordinatorV2Mock contract.

1. Click fundSubscription to fund the VRFV2Wrapper subscription. In this example, you can set the subid to 1 (which is your newly created subscription ID) and the amount to 10000000000000000000 (10 LINK).

1. A Metamask popup will open. Click on Confirm.

Deploy the VRF consumer contract

1. In the file explorer, open RandomNumberDirectFundingConsumerV2.sol.

1. Under DEPLOY & RUN TRANSACTIONS, select RandomNumberDirectFundingConsumerV2.

<ClickToZoom src="/images/vrf/mock/v2-deploydfconsumer.jpg" />

1. Under DEPLOY, fill in LINKADDRESS with the LinkToken contract address, and WRAPPERADDRESS with the deployed VRFV2Wrapper address.

1. Click on transact to deploy the RandomNumberDirectFundingConsumerV2 contract.

1. A Metamask popup will open. Click on Confirm.

1. Once deployed, you should see the RandomNumberDirectFundingConsumerV2 contract under Deployed Contracts.

<ClickToZoom src="/images/vrf/mock/v2-deployedddfconsumer.jpg" />

1. Note the address of the deployed contract.

Fund your VRF consumer contract

1. Under Deployed Contracts, open the functions list of your deployed LinkToken contract.

1. Click on transfer and fill in the to with your consumer contract address and value with LINK tokens amount. For this example, you can set the value to 10000000000000000000 (10 LINK).

<ClickToZoom src="/images/vrf/mock/v2-linktransfer.jpg" />

1. Click on transact.

1. A Metamask popup will open. Click on Confirm.

Request random words

Request three random words.

1. Under Deployed Contracts, open the functions list of your deployed RandomNumberConsumerV2 contract.

1. In requestRandomWords, fill in callbackGasLimit with 300000, requestConfirmations with 3 and numWords with 3.

<ClickToZoom src="/images/vrf/mock/v2-requestrandomwordsdof.jpg" />

1. Click on transact.

1. A Metamask popup will open.

<Aside type="note" title="Set your gas limit in MetaMask">

Remix IDE doesn't set the right gas limit, so you must edit the gas limit in MetaMask within the Advanced gas controls settings.

For this example to work, set the gas limit to 400,000 in MetaMask.

First, enable Advanced gas controls in your MetaMask settings.

Before confirming your transaction in MetaMask, navigate to the screen where you can edit the gas limit: Select Site suggested > Advanced > Advanced gas controls and select Edit next to the Gas limit amount. Update the Gas limit value to 400000 and select Save. Finally, confirm the transaction.

</Aside>

1. Click on Confirm.

1. In the RemixIDE console, read your transaction logs to find the VRF request ID. In this example, the request ID is 1.

<ClickToZoom src="/images/vrf/mock/v2-requestrandomwordlogsdf.jpg" />

1. Note your request ID.

Fulfill the VRF request

Because you are testing on a local blockchain environment, you must fulfill the VRF request yourself.

1. Under Deployed Contracts, open the functions list of your deployed VRFCoordinatorV2Mock contract.

1. Click fulfillRandomWords and fill in requestId with your VRF request ID and consumer with the VRFV2Wrapper contract address.

<ClickToZoom src="/images/vrf/mock/v2-fulfillrandomwordlogsdf.jpg" />

1. Click on transact.

1. A Metamask popup will open.

<Aside type="note" title="Set your gas limit in MetaMask">

Remix IDE doesn't set the right gas limit, so you must edit the gas limit in MetaMask within the Advanced gas controls settings.

For this example to work, set the gas limit to 1,000,000 in MetaMask.

First, enable Advanced gas controls in your MetaMask settings.

Before confirming your transaction in MetaMask, navigate to the screen where you can edit the gas limit: Select Site suggested > Advanced > Advanced gas controls and select Edit next to the Gas limit amount. Update the Gas limit value to 1000000 and select Save. Finally, confirm the transaction.

</Aside>

1. Click on Confirm.

1. In the RemixIDE console, read your transaction logs to find the random words.

<ClickToZoom src="/images/vrf/mock/v2-requestfulfilledlogsdf.jpg" />

Check the results

1. Under Deployed Contracts, open the functions list of your deployed RandomNumberDirectFundingConsumerV2 contract.

1. Click on lastRequestId to display the last request ID. In this example, the output is 1.

```
<ClickToZoom src="/images/vrf/mock/v2-deployedconsumerrequestiddf.jpg" />
```

1. Click on getRequestStatus with requestId equal to 1:

```
<ClickToZoom src="/images/vrf/mock/v2-deployedconsumerrequeststatusdf.jpg" />
```

1. You will get the amount paid, the status, and the random words.

```
<ClickToZoom src="/images/vrf/mock/v2-deployedconsumerrequeststatusresultsdf.jpg" />
```

Next steps

This guide demonstrated how to test a VRF v2 consumer contract on your local blockchain. We made the guide on RemixIDE for learning purposes, but you can reuse the same testing logic on another development environment, such as Truffle or Hardhat. For example, read the Hardhat Starter Kit RandomNumberDirectFundingConsumer unit tests.

index.mdx:

```
---
section: legacy
date: Last Modified
title: "Subscription Method"
isIndex: true
whatsnext:
  {
    "Get a Random Number": "/vrf/v2/subscription/examples/get-a-random-number",
    "Supported Networks": "/vrf/v2/subscription/supported-networks",
  }
metadata:
  title: "Generate Random Numbers for Smart Contracts using Chainlink VRF v2 - Subscription Method"
  description: "Learn how to securely generate random numbers for your smart contract with Chainlink VRF v2(an RNG). This guide uses the subscription method."
---
```

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
import { Aside, ClickToZoom } from "@components"
import { YouTube } from "@astro-community/astro-embed-youtube"
```

```
<Aside type="tip" title="VRF V2.5 Subscription Method">
  Refer to the VRF V2.5 Subscription Method Introduction page to learn how the subscription method works in VRF V2.5. To compare V2.5 and V2, refer to the migration guide.
</Aside>
```

```
<VrfCommon callout="subscription" />
```

This section explains how to generate random numbers using the subscription method.

<YouTube id="https://www.youtube.com/watch?v=rdJ5d8j1RCg" />

Subscriptions

VRF v2 requests receive funding from subscription accounts. The Subscription Manager lets you create an account and pre-pay for VRF v2, so you don't provide funding each time your application requests randomness. This reduces the total gas cost to use VRF v2. It also provides a simple way to fund your use of Chainlink products from a single location, so you don't have to manage multiple wallets across several different systems and applications.

<VrfCommon callout="ui" />

Subscriptions have the following core concepts:

- Subscription id: 64-bit unsigned integer representing the unique identifier of the subscription.
- Subscription accounts: An account that holds LINK tokens and makes them available to fund requests to Chainlink VRF v2 coordinators.
- Subscription owner: The wallet address that creates and manages a subscription account. Any account can add LINK to the subscription balance, but only the owner can add approved consuming contracts or withdraw funds.
- Consumers: Consuming contracts that are approved to use funding from your subscription account.
- Subscription balance: The amount of LINK maintained on your subscription account. Requests from consuming contracts will continue to be funded until the balance runs out, so be sure to maintain sufficient funds in your subscription balance to pay for the requests and keep your applications running.

For Chainlink VRF v2 to fulfill your requests, you must maintain a sufficient amount of LINK in your subscription balance. Gas cost calculation includes the following variables:

- Gas price: The current gas price, which fluctuates depending on network conditions.
- Callback gas: The amount of gas used for the callback request that returns your requested random values.
- Verification gas: The amount of gas used to verify randomness onchain.

The gas price depends on current network conditions. The callback gas depends on your callback function, and the number of random values in your request. The cost of each request is final only after the transaction is complete, but you define the limits you are willing to spend for the request with the following variables:

- Gas lane: The maximum gas price you are willing to pay for a request in wei. Define this limit by specifying the appropriate keyHash in your request. The limits of each gas lane are important for handling gas price spikes when Chainlink VRF bumps the gas price to fulfill your request quickly.
- Callback gas limit: Specifies the maximum amount of gas you are willing to spend on the callback request. Define this limit by specifying the callbackGasLimit value in your request.

Request and receive data

Requests to Chainlink VRF v2 follow the request and receive data cycle. This end-to-end diagram shows each step in the lifecycle of a VRF subscription request, and registering a smart contract with a VRF subscription account:

<ClickToZoom src="/images/vrf/v2-subscription-e2e.webp" />

Two types of accounts exist in the Ethereum ecosystem, and both are used in VRF:

- EOA (Externally Owned Account): An externally owned account that has a private key and can control a smart contract. Transactions can only be initiated by EOAs.
- Smart contract: A contract that does not have a private key and executes what it has been designed for as a decentralized application.

The Chainlink VRF v2 solution uses both offchain and onchain components:

- VRF v2 Coordinator (onchain component): A contract designed to interact with the VRF service. It emits an event when a request for randomness is made, and then verifies the random number and proof of how it was generated by the VRF service.
- VRF service (offchain component): Listens for requests by subscribing to the VRF Coordinator event logs and calculates a random number based on the block hash and nonce. The VRF service then sends a transaction to the VRFCoordinator including the random number and a proof of how it was generated.

Set up your contract and request

Set up your consuming contract:

1. Your contract must inherit VRFConsumerBaseV2.

1. Your contract must implement the fulfillRandomWords function, which is the callback VRF function. Here, you add logic to handle the random values after they are returned to your contract.

1. Submit your VRF request by calling requestRandomWords of the VRF Coordinator. Include the following parameters in your request:

- keyHash: Identifier that maps to a job and a private key on the VRF service and that represents a specified gas lane. If your request is urgent, specify a gas lane with a higher gas price limit. The configuration for your network can be found [here](#).
- ssubscriptionId: The subscription ID that the consuming contract is registered to. LINK funds are deducted from this subscription.
- requestConfirmations: The number of block confirmations the VRF service will wait to respond. The minimum and maximum confirmations for your network can be found [here](#).
- callbackGasLimit: The maximum amount of gas a user is willing to pay for completing the callback VRF function. Note that you cannot put a value larger than maxGasLimit of the VRF Coordinator contract (read coordinator contract limits for more details).
- numWords: The number of random numbers to request. The maximum random values that can be requested for your network can be found [here](#).

How VRF processes your request

After you submit your request, it is processed using the Request & Receive Data cycle. The VRF coordinator processes the request and determines the final charge to your subscription using the following steps:

1. The VRF coordinator emits an event.

1. The VRF service picks up the event and waits for the specified number of block confirmations to respond back to the VRF coordinator with the random values and a proof (requestConfirmations).

1. The VRF coordinator verifies the proof onchain, then it calls back the consuming contract fulfillRandomWords function.

Limits

Chainlink VRF v2 has some subscription limits and coordinator contract limits.

Subscription limits

Subscriptions are required to maintain a minimum balance, and they can support a limited number of consuming contracts.

Minimum subscription balance

Each subscription must maintain a minimum balance to fund requests from consuming contracts. This minimum balance requirement serves as a buffer against gas volatility by ensuring that all your requests have more than enough funding to go through. If your balance is below the minimum, your requests remain pending for up to 24 hours before they expire. After you add sufficient LINK to a subscription, pending requests automatically process as long as they have not expired.

In the Subscription Manager, the minimum subscription balance is displayed as the Max Cost, and it indicates the amount of LINK you need to add for a pending request to process. After the request is processed, only the amount actually consumed by the request is deducted from your balance. For example, if your minimum balance is 10 LINK, but your subscription balance is 5 LINK, you need to add at least 5 more LINK for your request to process. This does not mean that your request will ultimately cost 10 LINK. If the request ultimately costs 3 LINK after it has processed, then 3 LINK is deducted from your subscription balance.

The minimum subscription balance must be sufficient for each new consuming contract that you add to a subscription. For example, the minimum balance for a subscription that supports 20 consuming contracts needs to cover all the requests for all 20 contracts, while a subscription with one consuming contract only needs to cover that one contract.

For one request, the required size of the minimum balance depends on the gas lane and the size of the request. For example, a consuming contract that requests one random value will require a smaller minimum balance than a consuming contract that requests 50 random values. In general, you can estimate the required minimum LINK balance using the following formula where max verification gas is always 200,000 gwei.

$$\frac{((\text{Gas lane maximum (Max verification gas + Callback gas limit)}) / (1,000,000,000 \text{ Gwei/ETH}))}{(\text{ETH/LINK price})} + \text{LINK premium} = \text{Minimum LINK}$$

Here is the same formula, broken out into steps:

Gas lane maximum (Max verification gas + Callback gas limit) = Total estimated gas (Gwei)

Total estimated gas (Gwei) / 1,000,000,000 Gwei/ETH = Total estimated gas (ETH)

Total estimated gas (ETH) / (ETH/LINK price) = Total estimated gas (LINK)

Total estimated gas (LINK) + LINK premium = Minimum subscription balance (LINK)

Maximum consuming contracts

Each subscription supports up to 100 consuming contracts. If you need more than 100 consuming contracts, create multiple subscriptions.

Coordinator contract limits

You can see the configuration for each network on the Supported networks page.

You can also view the full configuration for each coordinator contract directly in Etherscan. As an example, view the Ethereum Mainnet VRF v2 coordinator contract configuration.

- Each coordinator has a MAXNUMWORDS parameter that limits the maximum number of random values you can receive in each request.
- Each coordinator has a maxGasLimit parameter, which is the maximum allowed callbackGasLimit value for your requests. You must specify a sufficient callbackGasLimit to fund the callback request to your consuming contract. This depends on the number of random values you request and how you process them in your fulfillRandomWords() function. If your callbackGasLimit is not sufficient, the callback fails but your subscription is still charged for the work done to generate your requested random values.

```
# supported-networks.mdx:
```

```
---
section: legacy
date: Last Modified
title: "Supported Networks - Subscription Method"
metadata:
  title: "Chainlink VRF v2 Supported Networks"
  linkToWallet: true
  image: "/files/OpenGraphV3.png"
---

import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"
import ResourcesCallout from
"@features/resources/callouts/ResourcesCallout.astro"
import { Address, Aside, CopyText } from "@components"

<Vrf25Common callout="supportednetworks" />
```

Chainlink VRF allows you to integrate provably fair and verifiably random data in your smart contract.

For implementation details, read [Introduction to Chainlink VRF](#).

Coordinator parameters

These parameters are configured in the coordinator contract. You can view these values by running getConfig on the coordinator or by viewing the coordinator contracts in a blockchain explorer.

- uint16 minimumRequestConfirmations: The minimum number of confirmation blocks on VRF requests before oracles respond
- uint32 maxGasLimit: The maximum gas limit supported for a fulfillRandomWords callback.
- uint32 stalenessSeconds: How long the coordinator waits until we consider the ETH/LINK price used for converting gas costs to LINK is stale and use fallbackWeiPerUnitLink
- uint32 gasAfterPaymentCalculation: How much gas is used outside of the payment calculation. This covers the additional operations required to decrement the subscription balance and increment the balance for the oracle that handled the request.

Fee parameters

Fee parameters are configured in the coordinator contract and specify the premium you pay per request in addition to the gas cost for the transaction. You can view them by running getFeeConfig on the coordinator. The uint32 fulfillMentFlatFeeLinkPPMTier1 parameter defines the fees per request specified in millionths of LINK.

The details for calculating the total transaction cost can be found [here](#).

Configurations

VRF v2 coordinators for subscription funding are available on several networks. To see a list of coordinators for direct funding, see the [Direct Funding Configurations](#) page.

<ResourcesCallout callout="bridgeRisks" />

Ethereum mainnet

<Vrf25Common callout="supportednetworks" />

Item	Value

LINK Token	<Address contractUrl="https://etherscan.io/token/0x514910771AF9Ca656af840dff83E8264EcF986CA" urlId="10x514910771AF9Ca656af840dff83E8264EcF986CA" urlClass="erc-token-address"/>
VRF Coordinator	<Address contractUrl="https://etherscan.io/address/0x271682DEB8C4E0901D1a1550aD2e64D568E69909" />
200 gwei Key Hash	<CopyText text="0x8af398995b04c28e9951adb9721ef74c74f93e6a478f39e7e0777be13527e7ef" code/>
500 gwei Key Hash	<CopyText text="0xff8dedfbfa60af186cf3c830acbc32c05aae823045ae5ea7da1e45fbfaba4f92" code/>
1000 gwei Key Hash	<CopyText text="0x9fe0eebf5e446e3c998ec9bb19951541aee00bb90ea201ae456421a2ded86805" code/>
Premium	0.25 LINK
Max Gas Limit	2,500,000
Minimum Confirmations	3
Maximum Confirmations	200
Maximum Random Values	500

Sepolia testnet

<Vrf25Common callout="supportednetworks" />

Testnet LINK and ETH are available from [faucets.chain.link](https://faucets.chain.link/sepolia).
Testnet ETH is also available from several public [ETH faucets](https://faucetlink.to/sepolia).

Item	Value

LINK Token	<Address

```
contractUrl="https://sepolia.etherscan.io/token/0x779877A7B0D9E8603169DdbD7836e478b4624789" urlId="111551110x779877A7B0D9E8603169DdbD7836e478b4624789"
urlClass="erc-token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://sepolia.etherscan.io/address/0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625" />
|
| 750 gwei Key Hash | <CopyText
text="0x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c" code/>
|
| Premium | 0.25 LINK
|
| Max Gas Limit | 2,500,000
|
| Minimum Confirmations | 3
|
| Maximum Confirmations | 200
|
| Maximum Random Values | 500
|
```

BNB Chain mainnet

```
<Vrf25Common callout="supportednetworks" />
```

```
<Aside type="tip" title="Important">
```

The LINK provided by the BNB Chain Bridge is not ERC-677 compatible, so cannot be used with Chainlink oracles. However, it can be converted to the official LINK token on BNB Chain using Chainlink's PegSwap service.

```
</Aside>
```

```
| Item | Value
| ----- |
|-----|
| LINK Token | <Address
contractUrl="https://bscscan.com/token/0x404460C6A5EdE2D891e8297795264fDe62ADBB75" urlId="560x404460C6A5EdE2D891e8297795264fDe62ADBB75" urlClass="erc-token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://bscscan.com/address/0xc587d9053cd1118f25F645F9E08BB98c9712A4EE" />
|
| 200 gwei Key Hash | <CopyText
text="0x114f3da0a805b6a67d6e9cd2ec746f7028f1b7376365af575cfea3550dd1aa04" code/>
|
| 500 gwei Key Hash | <CopyText
text="0xba6e730de88d94a5510ae6613898bfb0c3de5d16e609c5b7da808747125506f7" code/>
|
| 1000 gwei Key Hash | <CopyText
text="0x17cd473250a9a479dc7f234c64332ed4bc8af9e8ded7556aa6e66d83da49f470" code/>
|
| Premium | 0.005 LINK
|
| Max Gas Limit | 2,500,000
|
| Minimum Confirmations | 3
|
| Maximum Confirmations | 200
|
| Maximum Random Values | 500
|
```

|

BNB Chain testnet

<Vrf25Common callout="supportednetworks" />

Testnet LINK is available from <https://faucets.chain.link/bnb-chain-testnet>

Item	Value

LINK Token	<Address contractUrl="https://testnet.bscscan.com/address/0x84b9B910527Ad5C03A9Ca831909E21e236EA7b06" urlId="970x84b9B910527Ad5C03A9Ca831909E21e236EA7b06" urlClass="erc-token-address"/>
VRF Coordinator	<Address contractUrl="https://testnet.bscscan.com/address/0x6A2AAd07396B36Fe02a22b33cf443582f682c82f" />
50 gwei Key Hash	<CopyText text="0xd4bb89654db74673a187bd804519e65e3f71a52bc55f11da7601a13dcf505314" code/>
Premium	0.005 LINK
Max Gas Limit	2,500,000
Minimum Confirmations	3
Maximum Confirmations	200
Maximum Random Values	500

Polygon mainnet

<Vrf25Common callout="supportednetworks" />

<Aside type="tip" title="Important">

The LINK provided by the Polygon Bridge is not ERC-677 compatible, so cannot be used with Chainlink oracles. However, it can be converted to the official LINK token on Polygon using Chainlink's PegSwap service

</Aside>

Item	Value

LINK Token	<Address contractUrl="https://polygonscan.com/address/0xb0897686c545045aFc77CF20eC7A532E3120E0F1" urlId="1370xb0897686c545045aFc77CF20eC7A532E3120E0F1" urlClass="erc-token-address"/>
VRF Coordinator	<Address contractUrl="https://polygonscan.com/address/0xAE975071Be8F8eE67addBC1A82488F1C24858067" />
200 gwei Key Hash	<CopyText text="0x6e099d640cde6de9d40ac749b4b594126b0169747122711109c9985d47751f93" code/>

```
| 500 gwei Key Hash      | <CopyText  
text="0xcc294a196eeeb44da2888d17c0625cc88d70d9760a69d58d853ba6581a9ab0cd"  
code />
```

```
| 1000 gwei Key Hash     | <CopyText  
text="0xd729dc84e21ae57ffb6be0053bf2b0668aa2aaf300a2a7b2ddf7dc0bb6e875a8"  
code />
```

```
| Premium                | 0.0005 LINK  
| Max Gas Limit          | 2,500,000  
| Minimum Confirmations | 3  
| Maximum Confirmations | 200  
| Maximum Random Values | 500
```

Polygon Amoy testnet

<Vrf25Common callout="supportednetworks" />

```
| Item                    | Value  
| ----- |
```

```
----- |  
| LINK Token             | <Address  
contractUrl="https://amoy.polygonscan.com/address/0x0fd9e8d3af1aaee056eb9e802c3a762a667b1904" urlId="800020x0fd9e8d3af1aaee056eb9e802c3a762a667b1904" urlClass="erc-token-address"/> |  
| VRF Coordinator        | <Address  
contractUrl="https://amoy.polygonscan.com/address/0x7E10652Cb79Ba97bC1D0F38a1e8FaD8464a8a908" />
```

```
| 500 gwei Key Hash      | <CopyText  
text="0x3f631d5ec60a0ce16203bcd6aff7ffbc423e22e452786288e172d467354304c8"  
code />
```

```
| Premium                | 0.0005 LINK  
| Max Gas Limit          | 2,500,000  
| Minimum Confirmations | 3  
| Maximum Confirmations | 200  
| Maximum Random Values | 500
```

Avalanche mainnet

<Vrf25Common callout="supportednetworks" />

```
| Item                    | Value  
| ----- |
```

```
----- |  
| LINK Token             | <Address  
contractUrl="https://snowtrace.io/address/0x5947BB275c521040051D82396192181b4132
```

```
27A3" urlId="431140x5947BB275c521040051D82396192181b413227A3" urlClass="erc-
token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://snowtrace.io/address/0xd5D517aBE5cF79B7e95eC98dB0f0277788aF
F634" />
|
| 200 gwei Key Hash | <CopyText
text="0x83250c5584ffa93feb6ee082981c5ebe484c865196750b39835ad4f13780435d" code/>
|
| 500 gwei Key Hash | <CopyText
text="0x89630569c9567e43c4fe7b1633258df9f2531b62f2352fa721cf3162ee4ecb46" code/>
|
| 1000 gwei Key Hash | <CopyText
text="0x06eb0e2ea7cca202fc7c8258397a36f33d88568d2522b37aaa3b14ff6ee1b696" code/>
|
| Premium | 0.005 LINK
|
| Max Gas Limit | 2,500,000
|
| Minimum Confirmations | 1
|
| Maximum Confirmations | 200
|
| Maximum Random Values | 500
|
```

Avalanche Fuji testnet

<Vrf25Common callout="supportednetworks" />

Testnet LINK is available from <https://faucets.chain.link/fuji>

```
| Item | Value
| ----- |
|-----|
| LINK Token | <Address
contractUrl="https://testnet.snowtrace.io/address/0x0b9d5D9136855f6FEc3c0993feE6
E9CE8a297846" urlId="431130x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846"
urlClass="erc-token-address"/> |
|
| VRF Coordinator | <Address
contractUrl="https://testnet.snowtrace.io/address/0x2eD832Ba664535e5886b75D64C46
EB9a228C2610" />
|
| 300 gwei Key Hash | <CopyText
text="0x354d2f95da55398f44b7c7f77da56283d9c6c829a4bdf1bbcaf2ad6a4d081f61" code/>
|
| Premium | 0.005 LINK
|
| Max Gas Limit | 2,500,000
|
| Minimum Confirmations | 1
|
| Maximum Confirmations | 200
|
| Maximum Random Values | 500
|
```

Fantom mainnet

<Aside type="note" title="New Fantom subscriptions not supported">

Creating new Fantom subscriptions in the Subscription Manager is no longer supported in VRF V2 and is not supported in VRF V2.5. Existing Fantom subscriptions are still supported.

<Aside type="tip" title="Important">

You must use ERC-677 LINK on Fantom. ERC-20 LINK will not work with Chainlink services.

Item	Value

LINK Token	<Address contractUrl="https://ftmscan.com/token/0x6F43FF82CCA38001B6699a8AC47A2d0E66939407" urlId="2500x6F43FF82CCA38001B6699a8AC47A2d0E66939407" urlClass="erc-token-address"/>
VRF Coordinator	<Address contractUrl="https://ftmscan.com/address/0xd5D517aBE5cF79B7e95eC98dB0f0277788aFF634" />
4000 gwei Key Hash	<CopyText text="0xb4797e686f9a1548b9a2e8c68988d74788e0c4af5899020fb0c47784af76ddfa" code/>
10000 gwei Key Hash	<CopyText text="0x5881eea62f9876043df723cf89f0c2bb6f950da25e9dfe66995c24f919c8f8ab" code/>
20000 gwei Key Hash	<CopyText text="0x64ae04e5dba58bc08ba2d53eb33fe95bf71f5002789692fe78fb3778f16121c9" code/>
Premium	0.0005 LINK
Max Gas Limit	2,500,000
Minimum Confirmations	1
Maximum Confirmations	200
Maximum Random Values	500

Fantom testnet

Testnet LINK is available from <https://faucets.chain.link/fantom-testnet>

Item	Value

LINK Token	<Address contractUrl="https://testnet.ftmscan.com/address/0xfaFedb041c0DD4fA2Dc0d87a6B0979Ee6FA7af5F" urlId="40020xfaFedb041c0DD4fA2Dc0d87a6B0979Ee6FA7af5F" urlClass="erc-token-address"/>
VRF Coordinator	<Address contractUrl="https://testnet.ftmscan.com/address/0xbd13f08b8352A3635218ab9418E340c60d6Eb418" />

3000 gwei Key Hash	<CopyText text="0x121a143066e0f2f08b620784af77cccb35c6242460b4a8ee251b4b416abaebd4" code/>
Premium	0.0005 LINK
Max Gas Limit	2,500,000
Minimum Confirmations	1
Maximum Confirmations	200
Maximum Random Values	500

Arbitrum mainnet

<Vrf25Common callout="supportednetworks" />

Item	Value

LINK Token	<Address contractUrl="https://arbiscan.io/address/0xf97f4df75117a78c1A5a0DBb814Af92458539FB4" urlId="421610xf97f4df75117a78c1A5a0DBb814Af92458539FB4" urlClass="erc-token-address"/>
VRF Coordinator	<Address contractUrl="https://arbiscan.io/address/0x41034678D6C633D8a95c75e1138A360a28bA15d1" />
2 gwei Key Hash	<CopyText text="0x08ba8f62ff6c40a58877a106147661db43bc58dabfb814793847a839aa03367f" code/>
30 gwei Key Hash	<CopyText text="0x72d2b016bb5b62912afea355ebf33b91319f828738b111b723b78696b9847b63" code/>
150 gwei Key Hash	<CopyText text="0x68d24f9a037a649944964c2a1ebd0b2918f4a243d2a99701cc22b548cf2daff0" code/>
Premium	0.005 LINK
Max Gas Limit	2,500,000
Minimum Confirmations	1
Maximum Confirmations	200
Maximum Random Values	500

Arbitrum Sepolia testnet

<Vrf25Common callout="supportednetworks" />

Testnet LINK is available from <https://faucets.chain.link/arbitrum-sepolia>

Item	Value


```

----- |
| LINK Token | <Address
contractUrl="https://sepolia.arbiscan.io/address/0xb1D4538B4571d411F07960EF2838C
e337FE1E80E" urlId="4216140xb1D4538B4571d411F07960EF2838Ce337FE1E80E"
urlClass="erc-token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://sepolia.arbiscan.io/address/0x50d47e4142598E3411aA864e08a44
284e471AC6f" />
|
| 50 gwei Key Hash | <CopyText text="
0x027f94ff1465b3525f9fc03e9ff7d6d2c0953482246dd6ae07570c45d6631414" code/>
|
| Premium | 0.005 LINK
|
| Max Gas Limit | 2,500,000
|
| Minimum Confirmations | 1
|
| Maximum Confirmations | 200
|
| Maximum Random Values | 500
|

```

ui.mdx:

```

---
section: legacy
date: Last Modified
title: "Subscription Manager User Interface"
whatsnext:
  {
    "Security Considerations": "/vrf/v2/security",
    "Best Practices": "/vrf/v2/best-practices",
    "Migrating from VRF v1 to v2": "/vrf/v2/subscription/migration-from-v1",
    "Supported Networks": "/vrf/v2/subscription/supported-networks",
  }
metadata:
  title: "Subscription Manager User Interface"
  description: "Walkthrough Subscription Manager User Interface"
---

```

```

import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
import { Aside } from "@components"

```

```
<VrfCommon callout="subscription" />
```

The VRF Subscription Manager is available to help you create and manage VRF subscriptions. You can create and manage new V2.5 subscriptions, and manage existing V2 subscriptions, but you can no longer create new V2 subscriptions in the VRF Subscription Manager. Alternatively, you can create and manage V2 subscriptions programmatically.

This guide walks you through the main sections of the UI.

```

<Aside type="tip" title="Troubleshooting">
  Read the pending and failed requests sections to learn how to troubleshoot
  your VRF
  requests.
</Aside>

```

!VRF v2 UI overview

Subscription components:

- Status: Indicates if the subscription is still active or not.
- ID: The unique subscription identifier. Approved consuming contracts use LINK from this subscription to pay for each randomness request.
- Admin: The account address that owns this subscription ID.
- Consumers: The number of consuming contracts that are approved to make VRF requests using this subscription.
- Fulfillment: The number of successful randomness requests that are already completed.
- Balance: The amount of LINK remaining to be used for requests that use this subscription.

Actions menu

Expand the Actions menu to display actions you can take for any subscription that you own:

!VRF v2 UI with Actions menu expanded

- Fund subscription: Displays a field allowing you to fund your subscription from your connected wallet.
Specify how much LINK you'd like to send to your subscription.
- Add email address: Displays a field allowing you to add an email address to your subscription. This is an optional field.
- Cancel subscription: Displays a field allowing you to specify the account address to receive the remaining balance.
See the clean up instructions in the Get a Random Number guide to learn more.

Note: It is possible to fund subscriptions that you do not own. In that case, the Actions menu displays only the Fund subscription option.

Consumers

!VRF v2 UI consumers

The Consumers section lists the contracts that are allowed to use your subscription to pay for requests.

- Address: The address of the consuming contract.
- Added: The time when the consumer was added to the subscription.
- Last fulfillment: The last time a VRF request was fulfilled for the consumer.
- Total spent: The total amount of LINK that has been used by the consuming contract.

You can use this section to add or remove consumers.

Pending

!VRF v2 UI pending

The Pending list appears if there are requests currently being processed.

- Time: The time when the pending VRF request was made.
- Consumer: The address of the consuming contract.
- Transaction hash: The transaction hash of the pending VRF request.
- Status: A timer that informs you when the pending VRF request will move to a failed status. Note: Pending requests fail after 24h.
- Max Cost: The estimated upper limit of the transaction cost in LINK based on the configuration. Learn more about the minimum subscription balance.
- Projected Balance: This indicates when the subscription is underfunded and how many LINK tokens are required to fund the subscription.

History

Recent fulfillments

!VRF v2 UI recent fulfill

The Recent fulfillments tab shows the details for successful VRF fulfillments.

- Time: The time and block number indicating when the VRF request was successfully fulfilled.
- Consumer: The address of the consuming contract that initiated the VRF request.
- Transaction Hash: The transaction hash of the VRF callback.
- Status: The status of the request. Recent fulfillments always show Success.
- Spent: The total amount of LINK spent to fulfill the VRF request.
- Balance: The LINK balance of the subscription after the VRF request was fulfilled.

Events

!VRF v2 ui history events

The Events tab displays events linked to the subscription. There are five main events:

- Subscription created
- Subscription funded
- Consumer added
- Consumer removed
- Subscription canceled

Components of VRF events:

- Time: The time when the event happened.
- Event: The type of the event.
- Transaction Hash: The transaction hash for the event.
- Consumer: The address of the consuming contract. This is used only for Consumer added and Consumer canceled events.
- Amount:
 - For Subscription funded events, this indicates the amount of LINK added to the subscription balance.
 - For Subscription canceled events, this indicates the amount of LINK withdrawn from the subscription balance.
- Balance:
 - For Subscription funded events, this indicates the LINK balance of the subscription after it was funded.
 - For Subscription canceled events, this field should display 0.

Failed requests

!VRF v2 UI history failed

The Failed requests tab displays failed VRF requests.

- Time: The time when the VRF request was made.
- Transaction Hash: This can be either the transaction hash of the originating VRF request if the request was pending for over 24 hours or the transaction hash of the VRF callback if the callback failed.
- Status: The status of the request. Failed requests always show Failed.
- Reason: The reason why the request failed. Requests fail for one of the following reasons:
 - Pending for over 24 hours
 - Wrong key hash specified

- Callback gas limit set too low

```
# get-a-random-number.mdx:
```

```
---
section: legacy
date: Last Modified
title: "Get a Random Number with VRF V2"
whatsnext:
  {
    "Programmatic Subscription": "/vrf/v2/subscription/examples/programmatic-
subscription",
    "Subscription Manager UI": "/vrf/v2/subscription/ui",
    "Security Considerations": "/vrf/v2/security",
    "Best Practices": "/vrf/v2/best-practices",
    "Migrating from VRF v1 to v2": "/vrf/v2/subscription/migration-from-v1",
    "Supported Networks": "/vrf/v2/subscription/supported-networks",
  }
metadata:
  description: "How to generate a random number inside a smart contract using
Chainlink VRF."
---
```

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
import { Aside, CodeSample } from "@components"
```

```
<Aside type="tip" title="VRF V2.5 Subscription Tutorial">
  Refer to the VRF V2.5 version of this tutorial to learn how the
  subscription method works in VRF V2.5. To compare V2.5 and V2, refer to the
migration
  guide.
</Aside>
```

```
<VrfCommon callout="subscription" />
```

This guide explains how to get random values using a simple contract to request and receive random values from Chainlink VRF v2. For more advanced examples with programmatic subscription configuration, see the Programmatic Subscription page. To explore more applications of VRF, refer to our blog.

```
<VrfCommon callout="ui" />
```

Requirements

This guide assumes that you know how to create and deploy smart contracts on Ethereum testnets using the following tools:

- The Remix IDE
- MetaMask
- Sepolia testnet ETH

If you are new to developing smart contracts on Ethereum, see the Getting Started guide to learn the basics.

Create and fund a subscription

For this example, create a new subscription on the Sepolia testnet.

1. Open MetaMask and set it to use the Sepolia testnet. The Subscription Manager detects your network based on the active network in MetaMask.

1. Check MetaMask to make sure you have testnet ETH and LINK on Sepolia. You can get testnet ETH and LINK at faucets.chain.link.

1. Open the Subscription Manager at `vrf.chain.link`.

```
{/ prettier-ignore /}
```

```
<div class="remix-callout">
```

```
  <a href="https://vrf.chain.link">Open the Subscription Manager</a>
```

```
</div>
```

1. Click Create Subscription and follow the instructions to create a new subscription account. If you connect your wallet to the Subscription Manager, the Admin address for your subscription is prefilled and not editable. Optionally, you can enter an email address and a project name for your subscription, and both of these are private. MetaMask opens and asks you to confirm payment to create the account onchain. After you approve the transaction, the network confirms the creation of your subscription account onchain.

1. After the subscription is created, click Add funds and follow the instructions to fund your subscription.

- For your request to go through, you need to fund your subscription with enough LINK to meet your minimum subscription balance to serve as a buffer against gas volatility. For this example, a balance of 12 LINK is sufficient. (After your request is processed, it costs around 3 LINK, and that amount will be deducted from your subscription balance.)

- MetaMask opens to confirm the LINK transfer to your subscription. After you approve the transaction, the network confirms the transfer of your LINK token to your subscription account.

1. After you add funds, click Add consumer. A page opens with your account details and subscription ID.

1. Record your subscription ID, which you need for your consuming contract. You will add the consuming contract to your subscription later.

You can always find your subscription IDs, balances, and consumers at `vrf.chain.link`.

Now that you have a funded subscription account and your subscription ID, create and deploy a VRF v2 compatible contract.

Create and deploy a VRF v2 compatible contract

For this example, use the `VRFv2Consumer.sol` sample contract. This contract imports the following dependencies:

- `VRFConsumerBaseV2.sol(link)`
- `VRFCoordinatorV2Interface.sol(link)`
- `ConfirmedOwner.sol(link)`

The contract also includes pre-configured values for the necessary request parameters such as `vrfCoordinator` address, `gas lane keyHash`, `callbackGasLimit`, `requestConfirmations` and number of random words `numWords`. You can change these parameters if you want to experiment on different testnets, but for this example you only need to specify `subscriptionId` when you deploy the contract.

Build and deploy the contract on Sepolia.

1. Open the `VRFv2Consumer.sol` contract in Remix.

```
{/ prettier-ignore /}
```

```
<CodeSample src="samples/VRF/VRFv2Consumer.sol" showButtonOnly/>
```

1. On the Compile tab in Remix, compile the `VRFv2Consumer.sol` contract.

1. Configure your deployment. On the Deploy tab in Remix, select the Injected Provider environment, select the VRFv2Consumer contract from the contract list, and specify your subscriptionID so the constructor can set it.

!Example showing the deploy button with the subscriptionID field filled in Remix

1. Click the Deploy button to deploy your contract onchain. MetaMask opens and asks you to confirm the transaction.

1. After you deploy your contract, copy the address from the Deployed Contracts list in Remix. Before you can request randomness from VRF v2, you must add this address as an approved consuming contract on your subscription account.

!Example showing the contract address listed under the Contracts list in Remix

1. Open the Subscription Manager at vrf.chain.link and click the ID of your new subscription under the My Subscriptions list. The subscription details page opens.

1. Under the Consumers section, click Add consumer.

1. Enter the address of your consuming contract that you just deployed and click Add consumer. MetaMask opens and asks you to confirm the transaction.

Your example contract is deployed and approved to use your subscription balance to pay for VRF v2 requests. Next, request random values from Chainlink VRF.

Request random values

The deployed contract requests random values from Chainlink VRF, receives those values, builds a struct RequestStatus containing them and stores the struct in a mapping srequests. Run the requestRandomWords() function on your contract to start the request.

1. Return to Remix and view your deployed contract functions in the Deployed Contracts list.

1. Click the requestRandomWords() function to send the request for random values to Chainlink VRF. MetaMask opens and asks you to confirm the transaction. After you approve the transaction, Chainlink VRF processes your request. Chainlink VRF fulfills the request and returns the random values to your contract in a callback to the fulfillRandomWords() function. At this point, a new key requestId is added to the mapping srequests.

Depending on current testnet conditions, it might take a few minutes for the callback to return the requested random values to your contract. You can see a list of pending requests for your subscription ID at vrf.chain.link.

1. To fetch the request ID of your request, call lastRequestId().

1. After the oracle returns the random values to your contract, the mapping srequests is updated: The received random values are stored in srequests[requestId].randomWords.

1. Call getRequestStatus() specifying the requestId to display the random words.

You deployed a simple contract that can request and receive random values from Chainlink VRF. To see more advanced examples where the contract can complete the entire process including subscription setup and management, see the Programmatic Subscription page.

<Aside type="note" title="Note on Requesting Randomness">

Do not allow re-requesting or cancellation of randomness. For more information, see the VRF Security Considerations page.

</Aside>

Analyzing the contract

In this example, your MetaMask wallet is the subscription owner and you created a consuming contract to use that subscription. The consuming contract uses static configuration parameters.

<CodeSample src="samples/VRF/VRFv2Consumer.sol" />

The parameters define how your requests will be processed. You can find the values for your network in the Configuration page.

- uint64 sssubscriptionId: The subscription ID that this contract uses for funding requests.

- bytes32 keyHash: The gas lane key hash value, which is the maximum gas price you are willing to pay for a request in wei. It functions as an ID of the offchain VRF job that runs in response to requests.

- uint32 callbackGasLimit: The limit for how much gas to use for the callback request to your contract's fulfillRandomWords() function. It must be less than the maxGasLimit limit on the coordinator contract. Adjust this value for larger requests depending on how your fulfillRandomWords() function processes and stores the received random values. If your callbackGasLimit is not sufficient, the callback will fail and your subscription is still charged for the work done to generate your requested random values.

- uint16 requestConfirmations: How many confirmations the Chainlink node should wait before responding. The longer the node waits, the more secure the random value is. It must be greater than the minimumRequestBlockConfirmations limit on the coordinator contract.

- uint32 numWords: How many random values to request. If you can use several random values in a single callback, you can reduce the amount of gas that you spend per random value. The total cost of the callback request depends on how your fulfillRandomWords() function processes and stores the received random values, so adjust your callbackGasLimit accordingly.

The contract includes the following functions:

- requestRandomWords(): Takes your specified parameters and submits the request to the VRF coordinator contract.

- fulfillRandomWords(): Receives random values and stores them with your contract.

- getRequestStatus(): Retrieve request details for a given requestId.

<Aside type="note" title="Security Considerations">

Be sure to review your contracts to make sure they follow the best practices on the security considerations page.

</Aside>

Clean up

After you are done with this contract and the subscription, you can retrieve the remaining testnet LINK to use with other examples.

1. Open the Subscription Manager at vrf.chain.link and click the ID of your new subscription under the My Subscriptions list. The subscription details page opens.

1. On your subscription details page, expand the Actions menu and select Cancel subscription. A field displays, prompting you to add the wallet address you want to send the remaining funds to.

1. Enter your wallet address and click Cancel subscription. MetaMask opens and asks you to confirm the transaction. After you approve the transaction, Chainlink VRF closes your subscription account and sends the remaining LINK to your wallet.

Vyper example

You must import the VRFCoordinatorV2 Vyper interface. You can find it [here](#). You can find a VRFConsumerV2 example [here](#). Read the [apeworx-starter-kit README](#) to learn how to run the example.

```
# programmatic-subscription.mdx:
```

```
---
section: legacy
date: Last Modified
title: "Programmatic Subscription"
whatsnext:
  {
    "Subscription Manager UI": "/vrf/v2/subscription/ui",
    "Security Considerations": "/vrf/v2/security",
    "Best Practices": "/vrf/v2/best-practices",
    "Migrating from VRF v1 to v2": "/vrf/v2/subscription/migration-from-v1",
    "Supported Networks": "/vrf/v2/subscription/supported-networks",
  }
metadata:
  description: "Example contracts for generating a random number inside a smart
contract using Chainlink VRF v2."
---
```

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
import { Aside, CodeSample } from "@components"
```

```
<VrfCommon callout="subscription" />
```

How you manage the subscription depends on your randomness needs. You can configure your subscriptions using the Subscription Manager, but these examples demonstrate how to create your subscription and add your consumer contracts programmatically. For these examples, the contract owns and manages the subscription. Any wallet can provide funding to those subscriptions.

You can view and monitor your subscriptions in the Subscription Manager even if you create them programmatically. Go to vrf.chain.link to open the Subscription Manager.

Modifying subscriptions and configurations

Subscription configurations do not have to be static. You can change your subscription configuration dynamically by calling the following functions using the VRFCoordinatorV2Interface:

- Change the list of approved subscription consumers with:
 - `addConsumer(uint64 subId, address consumer).`
 - `removeConsumer(uint64 subId, address consumer).`
- Transfer the subscription ownership with:

- requestSubscriptionOwnerTransfer(uint64 subId, address newOwner).
- acceptSubscriptionOwnerTransfer(uint64 subId).
- View the subscription with getSubscription(uint64 subId).
- Cancel the subscription with cancelSubscription(uint64 subId).

To send LINK to the subscription balance, use the LINK token interface with `LINKTOKEN.transferAndCall(address(COORDINATOR), amount, abi.encode(subId))`. Any wallet can fund a subscription.

See the example in the Subscription manager contract section to learn how to create a contract that can change your subscription configuration.

Subscription manager contract

In this example, the contract operates as a subscription owner and can run functions to add consuming contracts to the subscription. The consuming contracts must include the `requestRandomWords()` function with the correct coordinator parameters and the correct subscription ID to request random values and use the subscription balance. The consuming contracts must also include the `fulfillRandomWords()` function to receive the random values.

Subscription owners and consumers do not have to be separate. This contract not only allows adding consumers with `addConsumer(address consumerAddress)` but can also act as a consumer by running its own `requestRandomWords()` function. This example contract includes a `createNewSubscription()` function in the `constructor()` that creates the subscription and adds itself as a consumer automatically when you deploy it.

```
<CodeSample src="samples/VRF/VRFv2SubscriptionManager.sol" />
```

To use this contract, compile and deploy it in Remix.

1. Open the contract in Remix.

1. Compile and deploy the contract using the Injected Provider environment. The contract includes all of the configuration variables that you need, but you can edit them if necessary. For a full list of available configuration variables, see the Supported Networks page.

This contract automatically creates a new subscription when you deploy it. Read the `ssubscriptionId` variable to find your subscription ID. You can use this value to find the subscription at `vrf.chain.link`.

1. In this example, the `topUpSubscription()` function sends LINK from your contract to the subscription. Fund your contract with at least three testnet LINK. Alternatively, you can send LINK directly to the subscription at `vrf.chain.link`. Any address can provide funding to a subscription balance. If you need testnet LINK, you can get it from `faucets.chain.link`.

1. Run the `topUpSubscription()` function to send LINK from your contract to your subscription balance. For this example, specify a value of `3000000000000000000`, which is equivalent to three LINK.

1. Run the `requestRandomWords()` function. The request might take several minutes to process. Track the pending request status at `vrf.chain.link`.

1. You can also add and test consumer contracts using the same programmatic subscription process:

1. Create and deploy a consumer contract that includes the following components:

- The `requestRandomWords()` function and the required variables and your subscription ID.

- The fulfillRandomWords() callback function.

You can use the example from the [Get a Random Number](#) guide.

1. After you deploy the consumer contract, add it to the subscription as an approved consumer using the addConsumer() function on your subscription manager contract. Specify the address of your consumer contract.

1. On the consumer contract, run the requestRandomWords() function to request and receive random values. The request might take several minutes to process. Track the pending request status at `vrf.chain.link`.

The consumer contract can continue to make requests until your subscription balance runs out. The subscription manager contract must maintain sufficient balance in the subscription so that the consumers can continue to operate.

1. If you need to remove consumer contracts from the subscription, use the removeConsumer() function. Specify the address of the consumer contract to be removed.

1. When you are done with your contracts and the subscription, run the cancelSubscription() function to close the subscription and send the remaining LINK to your wallet address. Specify the address of the receiving wallet.

Funding and requesting simultaneously

You can fund a subscription and request randomness in a single transaction. You must estimate how much the transaction might cost and determine the amount of funding to send to the subscription yourself. See the [Subscription billing](#) page to learn how to estimate request costs.

```
<CodeSample src="snippets/VRF/VRFv2FundAndRequestFunction.sol" />
```

Add this function to your contracts if you need to provide funding simultaneously with your requests. The transferAndCall() function sends LINK from your contract to the subscription, and the requestRandomWords() function requests the random words. Your contract still needs the fulfillRandomWords() callback function to receive the random values.

```
# test-locally.mdx:
```

```
---
```

```
section: legacy
```

```
title: "Local testing using a Mock contract"
```

```
metadata:
```

```
  description: "Example contract for generating random words using the VRF v2 subscription method on your local blockchain using a mock contract."
```

```
---
```

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
```

```
import ContentCommon from "@features/common/ContentCommon.astro"
```

```
import { CodeSample, ClickToZoom, Aside } from "@components"
```

```
<Aside type="tip" title="VRF V2.5 Subscription Mock Tutorial">
```

```
  Refer to the VRF V2.5 version of this subscription mock tutorial to learn how to test locally with VRF V2.5. To compare V2.5 and V2, refer to the migration guide.
```

```
</Aside>
```

```
<VrfCommon callout="subscription" />
```

This guide explains how to test Chainlink VRF v2 on a [Remix IDE](#) sandbox

blockchain environment. Note: You can reuse the same logic on another development environment, such as Hardhat or Truffle. For example, read the Hardhat Starter Kit RandomNumberConsumer unit tests.

<Aside type="caution" title="Test on public testnets thoroughly">

Even though local testing has several benefits, testing with a VRF mock covers the bare minimum of use cases. Make

sure to test your consumer contract thoroughly on public testnets.

</Aside>

Benefits of local testing

<ContentCommon section="localTestingBenefits" />

Testing logic

Complete the following tasks to test your VRF v2 consumer locally:

1. Deploy the VRFCoordinatorV2Mock. This contract is a mock of the VRFCoordinatorV2 contract.
1. Call the VRFCoordinatorV2Mock createSubscription function to create a new subscription.
1. Call the VRFCoordinatorV2Mock fundSubscription function to fund your newly created subscription. Note: You can fund with an arbitrary amount.
1. Deploy your VRF consumer contract.
1. Call the VRFCoordinatorV2Mock addConsumer function to add your consumer contract to your subscription.
1. Request random words from your consumer contract.
1. Call the VRFCoordinatorV2Mock fulfillRandomWords function to fulfill your consumer contract request.

Testing

Open the contracts on RemixIDE

Open VRFCoordinatorV2Mock and compile in Remix:

<CodeSample src="samples/VRF/mock/VRFCoordinatorV2Mock.sol" />

Open VRFv2Consumer and compile in Remix:

<CodeSample src="samples/VRF/mock/VRFv2Consumer.sol" />

Your RemixIDE file explorer should display VRFCoordinatorV2Mock.sol and VRFv2Consumer.sol:

<ClickToZoom src="/images/vrf/mock/v2-subscription-remix-fileexplorer.jpg" />

Deploy VRFCoordinatorV2Mock

1. Open VRFCoordinatorV2Mock.sol.
1. Under DEPLOY & RUN TRANSACTIONS, select VRFCoordinatorV2Mock.

<ClickToZoom src="/images/vrf/mock/v2-deploymock.jpg" />

1. Under DEPLOY, fill in the BASEFEE and GASPRICELINK. These variables are used in the VRFCoordinatorV2Mock contract to represent the base fee and the gas price (in LINK tokens) for the VRF requests. You can set: BASEFEE=1000000000000000000 and GASPRICELINK=10000000000.

1. Click on transact to deploy the VRFCoordinatorV2Mock contract.

1. Once deployed, you should see the VRFCoordinatorV2Mock contract under

Deployed Contracts.

```
<ClickToZoom src="/images/vrf/mock/v2-deployedmock.jpg" />
```

1. Note the address of the deployed contract.

Create and fund a subscription

1. Click on createSubscription to create a new subscription.

1. In the RemixIDE console, read your transaction decoded output to find the subscription ID. In this example, the subscription ID is 1.

```
<ClickToZoom src="/images/vrf/mock/v2-mocksubscription.jpg" />
```

1. Click on fundSubscription to fund your subscription. In this example, you can set the subid to 1 (which is your newly created subscription ID) and the amount to 10000000000000000000.

Deploy the VRF consumer contract

1. In the file explorer, open VRFv2Consumer.sol.

1. Under DEPLOY & RUN TRANSACTIONS, select RandomNumberConsumerV2.

```
<ClickToZoom src="/images/vrf/mock/v2-deployconsumer.jpg" />
```

1. Under DEPLOY, fill in SUBSCRIPTIONID with your subscription ID, vrfCoordinator with the deployed VRFCoordinatorV2Mock address and, KEYHASH with an arbitrary bytes32 (In this example, you can set the KEYHASH to 0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc).

1. Click on transact to deploy the RandomNumberConsumerV2 contract.

1. After the consumer contract is deployed, you should see the RandomNumberConsumerV2 contract under Deployed Contracts.

```
<ClickToZoom src="/images/vrf/mock/v2-deployedconsumer.jpg" />
```

1. Note the address of the deployed contract.

Add the consumer contract to your subscription

1. Under Deployed Contracts, open the functions list of your deployed VRFCoordinatorV2Mock contract.

1. Click on addConsumer and fill in the subid with your subscription ID and consumer with your deployed consumer contract address.

```
<ClickToZoom src="/images/vrf/mock/v2-addconsumer.jpg" />
```

1. Click on transact.

Request random words

1. Under Deployed Contracts, open the functions list of your deployed RandomNumberConsumerV2 contract.

1. Click on requestRandomWords.

```
<ClickToZoom src="/images/vrf/mock/v2-requestrandomwords.jpg" />
```

1. In the RemixIDE console, read your transaction logs to find the VRF request ID. In this example, the request ID is 1.

<ClickToZoom src="/images/vrf/mock/v2-requestrandomwordslogs.jpg" />

1. Note your request ID.

Fulfill the VRF request

Because you are testing on a local blockchain environment, you must fulfill the VRF request yourself.

1. Under Deployed Contracts, open the functions list of your deployed VRFCoordinatorV2Mock contract.

1. Click fulfillRandomWords and fill in requestId with your VRF request ID and consumer with your consumer contract address.

<ClickToZoom src="/images/vrf/mock/v2-fulfillrandomwords.jpg" />

1. Click on transact.

Check the results

1. Under Deployed Contracts, open the functions list of your deployed RandomNumberConsumerV2 contract.

1. Click on srequestId to display the last request ID. In this example, the output is 1.

<ClickToZoom src="/images/vrf/mock/v2-deployedconsumerrequestid.jpg" />

1. Each time you make a VRF request, your consumer contract requests two random words. After the request is fulfilled, the two random words are stored in the srandomWords array. You can check the stored random words by reading the two first indexes of the srandomWords array. To do so, click on the srandomWords function and:

1. Fill in the index with 0 then click on call to read the first random word.

{ " " }

<ClickToZoom src="/images/vrf/mock/v2-firstrandomword.jpg" />

1. Fill in the index with 1 then click on call to read the second random word.

{ " " }

<ClickToZoom src="/images/vrf/mock/v2-secondrandomword.jpg" />

Next steps

This guide demonstrated how to test a VRF v2 consumer contract on your local blockchain. We made the guide on RemixIDE for learning purposes, but you can reuse the same testing logic on another development environment, such as Truffle or Hardhat. For example, read the Hardhat Starter Kit RandomNumberConsumer unit tests.

arbitrum-cost-estimation.mdx:

section: vrf

date: Last Modified

title: "VRF Cost Estimation on Arbitrum"

```

metadata:
  title: "Estimating costs for Chainlink VRF v2.5 on Arbitrum"
  description: "Learn how to estimate costs for Chainlink VRF v2.5 on Arbitrum."
  isMdx: true
---
```

```
import { Tabs, TabsContent } from "@components/Tabs"
```

The total transaction costs for using Arbitrum involve both L2 gas costs and L1 costs. Arbitrum transactions are posted in batches to L1 Ethereum, which incurs an L1 cost. For an individual transaction, the total cost includes part of the L1 cost incurred to post the batch that included the transaction.

To learn how to estimate gas costs for Arbitrum, refer to the Arbitrum gas estimation tutorial and the full Arbitrum gas estimation script using their SDK. There is also a version of Arbitrum's gas estimation script extended to include VRF calculations.

Estimating Arbitrum gas costs with VRF

```

{/ prettier-ignore /}
<TabsContent sharedStore="vrfMethod" client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
```

VRF gas costs on L1 networks are calculated based on the amount of verification gas and callback gas used, multiplied by the gas price:

$$(\text{Gas price} \times (\text{Verification gas} + \text{Callback gas})) = \text{total gas cost}$$

For VRF Arbitrum transactions, add a buffer to estimate the additional L1 cost:

```

(L2GasPrice
  (Verification gas
    + Callback gas
    + L1 calldata gas buffer))) = total estimated gas cost
```

To calculate the L1 callback gas buffer:

- calldataSizeBytes: A static size for the transaction's calldata in bytes. Total: 720 bytes.
 - The amount Arbitrum adds to account for the static part of the transaction, fixed at 140 bytes.
 - The size of an ABI-encoded VRF V2.5 fulfillment, fixed at 580 bytes (fulfillmentTxSizeBytes).
- L1PricePerByte: The estimated cost of posting 1 byte of data on L1, which varies with L1 network gas prices.
- L2GasPrice: The L2 gas price, which varies with L2 network gas prices.

$$\begin{aligned} \text{L1 calldata gas buffer} &= (\text{calldataSizeBytes} \times \text{L1PricePerByte}) / \text{L2GasPrice} \\ &= (720 \times \text{L1PricePerByte}) / \text{L2GasPrice} \end{aligned}$$

This conversion allows us to estimate the L1 callback gas cost and incorporate it into the overall L2 gas estimate. You can add this estimated L1 callback gas buffer directly to the verification and callback gas:

```
(L2GasPrice
  (Verification gas
    + Callback gas
    + ((calldataSizeBytes  L1PricePerByte) / L2GasPrice)))) = total estimated
gas cost
```

</Fragment>

<Fragment slot="panel.2">

VRF gas costs on L1 networks are calculated based on the amount of coordinator overhead gas, wrapper overhead gas, and callback gas, multiplied by the gas price:

(Gas price (Coordinator overhead gas + Callback gas + Wrapper overhead gas)) = total gas cost

For VRF Arbitrum transactions, add a buffer to estimate the additional L1 cost:

```
(L2GasPrice
  (Coordinator overhead gas
    + Callback gas
    + Wrapper overhead gas
    + L1 calldata gas buffer))) = total estimated gas cost
```

To calculate the L1 callback gas buffer:

- calldataSizeBytes: A static size for the transaction's calldata in bytes. Total: 720 bytes.
- The amount Arbitrum adds to account for the static part of the transaction, fixed at 140 bytes.
- The size of an ABI-encoded VRF V2.5 fulfillment, fixed at 580 bytes (fulfillmentTxSizeBytes).
- L1PricePerByte: The estimated cost of posting 1 byte of data on L1, which varies with L1 network gas prices.
- L2GasPrice: The L2 gas price, which varies with L2 network gas prices.

L1 calldata gas buffer = (calldataSizeBytes L1PricePerByte) / L2GasPrice

= (720 L1PricePerByte) / L2GasPrice

This conversion allows us to estimate the L1 callback gas cost and incorporate it into the overall L2 gas estimate. You can add this estimated L1 callback gas buffer directly to the other gas figures:

```
(L2GasPrice
  (Coordinator overhead gas
    + Callback gas
    + Wrapper overhead gas
    + ((calldataSizeBytes  L1PricePerByte) / L2GasPrice)))) = total estimated
gas cost
```

</Fragment>

</TabsContent>

Arbitrum and VRF gas estimation code

This sample extends the original Arbitrum gas estimation script to estimate gas costs for VRF subscription and direct funding requests on Arbitrum.

The following snippet shows only the VRF variables and calculations that were added to the Arbitrum gas estimation script. To learn more about how Arbitrum gas is calculated, refer to the Arbitrum gas estimation tutorial. To run this code, use the full Arbitrum gas estimation script that includes VRF calculations.

```

typescript
// VRF variables and calculations
// -----
// Full script:
https://github.com/smartcontractkit/smart-contract-examples/tree/main/vrf-arbitrum-gas-estimation
// -----
// Estimated upper bound of verification gas for VRF subscription.
// To see an estimate with an average amount of verification gas,
// adjust this to 115000.
const maxVerificationGas = 200000

// The L1 Calldata size includes:
// Arbitrum's static 140 bytes for transaction metadata
// VRF V2's static 580 bytes, the size of a fulfillment's calldata abi-encoded in bytes
// (from sfulfillmentTxSizeBytes in VRFV2Wrapper.sol)
const VRFCallDataSizeBytes = 140 + 580

// For direct funding only
// Coordinator gas is verification gas.
// Some overhead gas values vary by network. These are hardcoded for Sepolia.
// Refer to https://docs.chain.link/vrf/v2-5/supported-networks to find values for other networks.
const wrapperGasOverhead = 13400
const coordinatorGasOverheadNative = 90000
const coordinatorGasOverheadLink = 112000
const coordinatorGasOverheadPerWord = 435

// VRF user settings
const callbackGasLimit = 175000
const numWords = 2 // Max 10 per request

// Estimate VRF L1 buffer
const VRFL1CostEstimate = L1P.mul(VRFCallDataSizeBytes)
const VRFL1Buffer = VRFL1CostEstimate.div(P)

// VRF Subscription gas estimate
// L2 gas price (P) (maxVerificationGas + callbackGasLimit + VRFL1Buffer)
const VRFL2SubscriptionGasSubtotal = BigNumber.from(maxVerificationGas + callbackGasLimit)
const VRFSubscriptionGasTotal = VRFL2SubscriptionGasSubtotal.add(VRFL1Buffer)
const VRFSubscriptionGasEstimate = P.mul(VRFSubscriptionGasTotal)

// VRF Direct funding gas estimate
// L2 gas price (P) (coordinatorGasOverheadLink + callbackGasLimit + wrapperGasOverhead + VRFL1Buffer)
// If using native tokens, change coordinatorGasOverheadLink to coordinatorGasOverheadNative below
const directFundingGasOverheadPerWord = coordinatorGasOverheadPerWord.mul(numWords)
const VRFL2DirectFundingGasSubtotal = BigNumber.from(
  coordinatorGasOverheadLink + wrapperGasOverhead + callbackGasLimit + directFundingGasOverheadPerWord
)

```



```
const VRFDirectFundingGasTotal = VRFL2DirectFundingGasSubtotal.add(VRFL1Buffer)
const VRFDirectFundingGasEstimate = P.mul(VRFDirectFundingGasTotal)
```

best-practices.mdx:

```
---
section: vrf
date: Last Modified
title: "VRF Best Practices"
metadata:
  title: "Chainlink VRF API Reference"
  description: "Best practices for using Chainlink VRF."
---
```

```
import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"
import { CodeSample } from "@components"
```

```
<Vrf25Common callout="security" />
```

These are example best practices for using Chainlink VRF. To explore more applications of VRF, refer to our blog.

Getting a random number within a range

If you need to generate a random number within a given range, use modulo to define the limits of your range. Below you can see how to get a random number in a range from 1 to 50.

```
{/ prettier-ignore /}
solidity
function fulfillRandomWords(
  uint256, / requestId /
  uint256[] memory randomWords
) internal override {
  // Assuming only one random word was requested.
  srandomRange = (randomWords[0] % 50) + 1;
}
```

Getting multiple random values

If you want to get multiple random values from a single VRF request, you can request this directly with the numWords argument:

- If you are using the VRF v2.5 subscription method, see the full example code for an example where one request returns multiple random values.

Processing simultaneous VRF requests

If you want to have multiple VRF requests processing simultaneously, create a mapping between requestId and the response. You might also create a mapping between the requestId and the address of the requester to track which address made each request.

```
{/ prettier-ignore /}
solidity
mapping(uint256 => uint256[]) public srequestIdToRandomWords;
mapping(uint256 => address) public srequestIdToAddress;
uint256 public srequestId;

function requestRandomWords() external onlyOwner returns (uint256) {
  uint256 requestId = svrfCoordinator.requestRandomWords(
```

```

        VRFV2PlusClient.RandomWordsRequest({
            keyHash: keyHash,
            subId: svrfSubscriptionId,
            requestConfirmations: requestConfirmations,
            callbackGasLimit: callbackGasLimit,
            numWords: numWords,
            extraArgs:
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment:
true})) // new parameter
        })
    );
    srequestIdToAddress[requestId] = msg.sender;

    // Store the latest requestId for this example.
    srequestId = requestId;

    // Return the requestId to the requester.
    return requestId;
}

function fulfillRandomWords(
    uint256 requestId,
    uint256[] memory randomWords
) internal override {
    // You can return the value to the requester,
    // but this example simply stores it.
    srequestIdToRandomWords[requestId] = randomWords;
}

```

You could also map the requestId to an index to keep track of the order in which a request was made.

```

{/ prettier-ignore /}
solidity
mapping(uint256 => uint256) srequestIdToRequestIndex;
mapping(uint256 => uint256[]) public srequestIndexToRandomWords;
uint256 public requestCounter;

function requestRandomWords() external onlyOwner {
    uint256 requestId = svrfCoordinator.requestRandomWords(
        VRFV2PlusClient.RandomWordsRequest({
            keyHash: keyHash,
            subId: svrfSubscriptionId,
            requestConfirmations: requestConfirmations,
            callbackGasLimit: callbackGasLimit,
            numWords: numWords,
            extraArgs:
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment:
true})) // new parameter
        })
    );
    srequestIdToRequestIndex[requestId] = requestCounter;
    requestCounter += 1;
}

function fulfillRandomWords(
    uint256 requestId,
    uint256[] memory randomWords
) internal override {
    uint256 requestNumber = srequestIdToRequestIndex[requestId];
    srequestIndexToRandomWords[requestNumber] = randomWords;
}

```

Processing VRF responses through different execution paths

If you want to process VRF responses depending on predetermined conditions, you can create an enum. When requesting for randomness, map each requestId to an enum. This way, you can handle different execution paths in fulfillRandomWords. See the following example:

```
{/ prettier-ignore /}
solidity
// SPDX-License-Identifier: MIT
// An example of a consumer contract that relies on a subscription for funding.
// It shows how to setup multiple execution paths for handling a response.
pragma solidity 0.8.19;

import {LinkTokenInterface} from
"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.sol";
import {IVRFCoordinatorV2Plus} from
"@chainlink/contracts/src/v0.8/vrf/dev/interfaces/IVRFCoordinatorV2Plus.sol";
import {VRFConsumerBaseV2Plus} from
"@chainlink/contracts/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";
import {VRFV2PlusClient} from
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";

/
THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY.
THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE.
DO NOT USE THIS CODE IN PRODUCTION.
/

contract VRFv2MultiplePaths is VRFConsumerBaseV2Plus {

    // Your subscription ID.
    uint256 ssubscriptionId;

    // Sepolia coordinator. For other networks,
    // see https://docs.chain.link/docs/vrf/v2-5/supported-networks
    address vrfCoordinatorV2Plus = 0x9DdfaCa8183c41ad55329BdeeD9F6A8d53168B1B;

    // The gas lane to use, which specifies the maximum gas price to bump to.
    // For a list of available gas lanes on each network,
    // see https://docs.chain.link/docs/vrf/v2-5/supported-networks
    bytes32 keyHash =
        0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae;

    uint32 callbackGasLimit = 100000;

    // The default is 3, but you can set this higher.
    uint16 requestConfirmations = 3;

    // For this example, retrieve 1 random value in one request.
    // Cannot exceed VRFCoordinatorV25.MAXNUMWORDS.
    uint32 numWords = 1;

    enum Variable {
        A,
        B,
        C
    }

    uint256 public variableA;
    uint256 public variableB;
    uint256 public variableC;
```

```

mapping(uint256 => Variable) public requests;

// events
event FulfilledA(uint256 requestId, uint256 value);
event FulfilledB(uint256 requestId, uint256 value);
event FulfilledC(uint256 requestId, uint256 value);

constructor(uint256 subscriptionId)
VRFConsumerBaseV2Plus(vrfCoordinatorV2Plus) {
    svrfCoordinator = IVRFCoordinatorV2Plus(vrfCoordinatorV2Plus);
    ssubscriptionId = subscriptionId;
}

function updateVariable(uint256 input) public {
    uint256 requestId =
svrfCoordinator.requestRandomWords(VRFV2PlusClient.RandomWordsRequest({
    keyHash: keyHash,
    subId: ssubscriptionId,
    requestConfirmations: requestConfirmations,
    callbackGasLimit: callbackGasLimit,
    numWords: numWords,
    extraArgs:
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: true}))
    })
    );

    if (input % 2 == 0) {
        requests[requestId] = Variable.A;
    } else if (input % 3 == 0) {
        requests[requestId] = Variable.B;
    } else {
        requests[requestId] = Variable.C;
    }
}

function fulfillRandomWords(
    uint256 requestId,
    uint256[] memory randomWords
) internal override {
    Variable variable = requests[requestId];
    if (variable == Variable.A) {
        fulfillA(requestId, randomWords[0]);
    } else if (variable == Variable.B) {
        fulfillB(requestId, randomWords[0]);
    } else if (variable == Variable.C) {
        fulfillC(requestId, randomWords[0]);
    }
}

function fulfillA(uint256 requestId, uint256 randomWord) private {
    // execution path A
    variableA = randomWord;
    emit FulfilledA(requestId, randomWord);
}

function fulfillB(uint256 requestId, uint256 randomWord) private {
    // execution path B
    variableB = randomWord;
    emit FulfilledB(requestId, randomWord);
}

function fulfillC(uint256 requestId, uint256 randomWord) private {
    // execution path C
    variableC = randomWord;
}

```

```

    emit FulfilledC(requestId, randomWord);
  }
}

```

billing.mdx:

```

---
section: vrf
date: Last Modified
title: "VRF Billing"
metadata:
  title: "Estimating costs for Chainlink VRF v2.5"
  description: "Learn how to estimate costs for Chainlink VRF v2.5."
isMdx: true
---

```

```

import { CodeSample } from "@components"
import { TabsContent } from "@components/Tabs"

```

This guide explains how to estimate VRF 2.5 costs for both the subscription and direct funding methods.

Understanding transaction costs

```

{/ prettier-ignore /}
<TabsContent sharedStore="vrfMethod" client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
For Chainlink VRF v2.5 to fulfill your requests, you must maintain enough funds
in your subscription balance. Gas cost calculation includes the following
variables:

```

- Gas price: The current gas price, which fluctuates depending on network conditions.
- Callback gas: The amount of gas used for the callback request that returns your requested random values.
- Verification gas: The amount of gas used to verify randomness onchain.

The gas price depends on current network conditions. The callback gas depends on your callback function, and the number of random values in your request. The cost of each request is final only after the transaction is complete, but you define the limits you are willing to spend for the request with the following variables:

- Gas lane: The maximum gas price you are willing to pay for a request in wei. Define this limit by specifying the appropriate keyHash in your request. The limits of each gas lane are important for handling gas price spikes when Chainlink VRF bumps the gas price to fulfill your request quickly.
- Callback gas limit: Specifies the maximum amount of gas you are willing to spend on the callback request. Define this limit by specifying the callbackGasLimit value in your request.

```

</Fragment>

```

```

<Fragment slot="panel.2">
For Chainlink VRF v2.5 to fulfill your requests, you must maintain enough funds
in your consuming contract.

```

Gas cost calculation includes the following variables:

- Callback gas: The amount of gas used for the callback request that returns your requested random values. The callback gas depends on your callback function and the number of random values in your request. Set the callback gas limit to specify the maximum amount of gas you are willing to spend on the callback request.
- Number of random values requested: The number of random values (numWords) per request to VRF.
- Gas price: The current gas price, which fluctuates depending on network conditions.
- Coordinator overhead gas: The amount of gas used to verify randomness onchain. This consists of two components:
 - Coordinator overhead gas (Native or LINK): The coordinator overhead gas has different values depending on whether you're using LINK or native tokens.
 - Coordinator overhead gas per word: The amount of additional gas the coordinator uses per random value ("word") that you request.
- Wrapper overhead gas: The amount of gas used by the VRF Wrapper contract.

Because the consuming contract directly pays for the request, the cost is calculated during the request and not during the callback when the randomness is fulfilled. Test your callback function to learn how to correctly estimate the callback gas limit.

- If the gas limit is underestimated, the callback fails and the consuming contract is still charged for the work done to generate the requested random values.
- If the gas limit is overestimated, the callback function will be executed but your contract is not refunded for the excess gas amount that you paid.

Make sure that your consuming contracts have enough funds in either LINK or native tokens to cover the transaction costs. If the consuming contract doesn't have enough funds, your request will revert.

</Fragment>
</TabsContent>

Estimate gas costs

```
{/ prettier-ignore /}  
<TabsContent sharedStore="vrfMethod" client:visible>  
<Fragment slot="tab.1">Subscription</Fragment>  
<Fragment slot="tab.2">Direct funding</Fragment>  
<Fragment slot="panel.1">  
You need to pre-fund your subscription enough to meet the minimum subscription  
balance  
in order to have a buffer against gas volatility.
```

After the request is complete, the final gas cost is recorded based on how much gas is used for the verification and callback.
The actual cost of the request is deducted from your subscription balance.

The total gas cost in wei for your request uses the following formula:

$(\text{Gas price} \times (\text{Verification gas} + \text{Callback gas})) = \text{total gas cost}$

If you're paying for VRF in LINK, the total gas cost is converted to LINK using

the ETH/LINK data feed. In the unlikely event that the data feed is unavailable, the VRF coordinator uses the fallbackWeiPerUnitLink value for the conversion instead. The fallbackWeiPerUnitLink value is defined in the coordinator contract for your selected network.

</Fragment>

<Fragment slot="panel.2">

The final gas cost to fulfill randomness is estimated based on how much gas is expected for the verification and callback. The total gas cost in wei uses the following formula:

$$(\text{Gas price} \quad (\text{Coordinator gas overhead} \\ + \text{Callback gas limit} \\ + \text{Wrapper gas overhead})) = \text{total gas cost}$$

The total gas cost is converted to LINK using the ETH/LINK data feed. In the unlikely event that the data feed is unavailable, the VRF Wrapper uses the fallbackWeiPerUnitLink value for the conversion instead. The fallbackWeiPerUnitLink value is defined in the VRF v2.5 Wrapper contract for your selected network.

The maximum allowed callbackGasLimit value for your requests is defined in the Coordinator contract supported networks page. Because the VRF v2.5 Wrapper adds gas overheads, your callbackGasLimit must not exceed $\text{maxGasLimit} - \text{wrapperGasOverhead}$.

</Fragment>

</TabsContent>

Apply premium

{/ prettier-ignore /}

<TabsContent sharedStore="vrfMethod" client:visible>

<Fragment slot="tab.1">Subscription</Fragment>

<Fragment slot="tab.2">Direct funding</Fragment>

<Fragment slot="panel.1">

The premium is charged as a percentage of the overall gas cost. The premium is defined in the coordinator contract. Premium percentages are listed there as whole integers. For example, a 20% premium is listed as 20.

$$(\text{total gas cost}) \quad ((100 + \text{Premium percentage}) / 100) = \text{total request cost}$$

The total request cost is charged to your subscription balance. Since you have the option to pay for VRF requests either in LINK or the native token for the network you're using, your subscription can have both a LINK balance and a native token balance. The premium is higher when you pay with native tokens than when you pay with LINK. For example, the premium percentage for using Ethereum is 24 if you pay with Ethereum, and 20 if you pay with LINK.

</Fragment>

<Fragment slot="panel.2">

The premium is divided in two parts:

- Wrapper premium: This premium is charged as a percentage of the overall gas cost. Premium percentages are listed there as whole integers. For example, a 20% premium is listed as 20. You can find the percentage for your network in the Supported networks page.
- Coordinator premium: A flat fee. This premium is defined in the fulfillmentFlatFeeLinkPPMTier1 parameter in millionths of LINK. You can find the

flat fee of the coordinator for your network in the Supported networks page.

(Coordinator premium
+ (total gas cost) ((100 + Premium percentage) / 100)) = total request cost

The total request cost is charged to your consuming contract. The premium is higher when you pay with native tokens than when you pay with LINK. For example, the premium percentage for using Ethereum is 24 if you pay with Ethereum, and 20 if you pay with LINK.

</Fragment>
</TabsContent>

Subscription cost examples

These are example calculations of a VRF subscription request on Ethereum, shown in both ETH and LINK. The values for other supported networks are available on the Supported Networks page. The examples show how to estimate the following:

- The minimum subscription balance, which is a higher amount you need to reserve before your request is processed. This provides a buffer in case gas prices go higher when processing the request. The VRF Subscription Manager displays your minimum subscription balance as Max Cost.
- The actual cost of the request after it is processed, which is lower than the minimum subscription balance.

Estimate minimum subscription balance

These example calculations show an estimated minimum subscription balance for using VRF on Ethereum, shown in both ETH and LINK. The premium is higher when you pay with native tokens than when you pay with LINK.

<TabsContent sharedStore="feePaymentMethod" client:visible>
<Fragment slot="tab.1">Paying in LINK</Fragment>
<Fragment slot="tab.2">Paying in ETH</Fragment>
<Fragment slot="panel.1">

Parameter	Value
-----	-----
Gas lane	500 gwei
Callback gas limit	100000
Max verification gas	200000
Premium percentage	20

1. Calculate the total gas cost, using the maximum possible gas price for the selected gas lane, the estimated maximum verification gas, and the full callback gas limit:

Gas cost calculation	Total gas cost
-----	-----
Gas price x (Verification gas + Callback gas)	
500 gwei x (200000 + 100000)	150000000 gwei (0.15 ETH)

1. Apply the premium percentage to get the total maximum cost of a request:

Applying premium percentage	Maximum request cost (ETH)
-----	-----
Total gas cost (ETH) \ ((100 + premium percentage)/100)	

0.15 ETH \ ((100 + 20)/100)	0.18 ETH
-----------------------------	----------

1. Convert the total cost to LINK using the LINK/ETH feed.

For this example, assume the feed returns a conversion value of 0.005 ETH per 1 LINK.

ETH to LINK cost conversion Maximum request cost (LINK)
----- -----
0.18 ETH / 0.005 ETH/LINK 36 LINK

For this example request to go through, you need to reserve a minimum subscription balance of 36 LINK, but that does not mean the actual request will cost 36 LINK. Check the Max Cost in the Subscription Manager to view the minimum subscription balance for all your contracts. When your request is processed, the actual cost of the request is calculated and deducted from your subscription balance. See the next section for an example of how to calculate the actual request cost.

</Fragment>

<Fragment slot="panel.2">

Parameter	Value
-----	-----
Gas lane	500 gwei
Callback gas limit	100000
Max verification gas	200000
Premium percentage	24

1. Calculate the total gas cost, using the maximum possible gas price for the selected gas lane, the estimated maximum verification gas, and the full callback gas limit:

Gas cost calculation	Total gas cost
-----	-----
Gas price x (Verification gas + Callback gas)	
500 gwei x (200000 + 100000)	150000000 gwei (0.15 ETH)

1. Apply the premium percentage to get the total maximum cost of a request:

Applying premium percentage	Maximum request
cost (ETH)	

Total gas cost (ETH) \ ((100 + premium percentage)/100)	
0.15 ETH \ ((100 + 24)/100)	0.186 ETH

For this example request to go through, you need to reserve a minimum subscription balance of 0.186 ETH, but that does not mean the actual request will cost 0.186 ETH. Check the Max Cost in the Subscription Manager to view the minimum subscription balance for all your contracts. When your request is processed, the actual cost of the request is deducted from your subscription balance. See the next section for an example of how to calculate the actual request cost.

</Fragment>

</TabsContent>

Estimate VRF request cost

These example calculations show a cost breakdown of a VRF subscription request on the Ethereum network. Check Etherscan for current gas prices.

```
<TabsContent sharedStore="feePaymentMethod" client:visible>
<Fragment slot="tab.1">Paying in LINK</Fragment>
<Fragment slot="tab.2">Paying in ETH</Fragment>
<Fragment slot="panel.1">
```

Parameter	Value
-----	-----
Actual gas price	50 gwei
Callback gas used	95000
Verification gas used	115000
Premium percentage	20

1. Calculate the total gas cost:

Gas cost calculation	Total gas cost
-----	-----
Gas price x (Verification gas + Callback gas)	
50 gwei x (115000 + 95000)	10500000 gwei (0.0105 ETH)

1. Apply the premium percentage to get the total cost of a request:

Applying premium percentage	Total request
cost (ETH)	
-----	-----
Total gas cost (ETH) \ ((100 + premium percentage)/100)	
0.0105 ETH \ ((100 + 20)/100)	0.0126 ETH

1. Convert the total cost to LINK using the LINK/ETH feed.

For this example, assume the feed returns a conversion value of ̂ 0.005 ETH per 1 LINK.

ETH to LINK cost conversion	Total gas cost (LINK)
-----	-----
0.0126 ETH / 0.005 ETH/LINK	2.52 LINK

This example request would cost 2.52 LINK, which is deducted from your subscription balance.

```
</Fragment>
```

```
<Fragment slot="panel.2">
```

Parameter	Value
-----	-----
Actual gas price	50 gwei
Callback gas used	95000
Verification gas used	115000
Premium percentage	24

1. Calculate the total gas cost:

Gas cost calculation	Total gas cost
-----	-----
Gas price x (Verification gas + Callback gas)	
50 gwei x (115000 + 95000)	10500000 gwei (0.0105 ETH)

1. Apply the premium percentage to get the total maximum cost of a request:

Applying premium percentage	Maximum request
cost (ETH)	

Total gas cost (ETH) \ ((100 + premium percentage)/100)	
0.0105 ETH \ ((100 + 24)/100)	0.01302 ETH

This example request would cost 0.01302 ETH, which is deducted from your subscription balance.

</Fragment>
</TabsContent>

Direct funding cost examples

These are example calculations of a VRF direct funding request on Ethereum, shown in both ETH and LINK. The values for other supported networks are available on the Supported Networks page.

<TabsContent sharedStore="feePaymentMethod" client:visible>
<Fragment slot="tab.1">Paying in LINK</Fragment>
<Fragment slot="tab.2">Paying in ETH</Fragment>
<Fragment slot="panel.1">

Parameter	Value
-----	-----
Gas price	50 gwei
Callback gas limit	100000
Coordinator gas overhead (LINK)	112000
Wrapper gas overhead	13400
Coordinator gas overhead per word	435
Number of random values (words)	2
Wrapper premium percentage	20

1. Calculate the total gas cost:

Gas cost calculation
Total gas cost

Gas price (Coordinator overhead gas + Callback gas limit + Wrapper gas overhead + (Coordinator overhead gas per word Number of words))
50 gwei x (112000 + 100000 + 13400 + (435 \ 2))
11313500 gwei (0.0113135 ETH)

1. Convert the gas cost to LINK using the LINK/ETH feed.

For this example, assume the feed returns a conversion value of 1 0.004 ETH per 1 LINK.

ETH to LINK cost conversion	Total gas cost (LINK)
-----	-----
0.0113135 ETH / 0.004 ETH/LINK	2.828375 LINK

1. Apply the premium percentage to get the total cost of a request:

Applying premium percentage	Request cost
(LINK)	

```

| ----- |
|-----|
| Total gas cost (LINK) \ ((100 + premium percentage)/100) |
|
| (2.828375 LINK \ (100 + 20))/100 | 3.39405 LINK
|

```

This example request would cost 3.39405 LINK.

</Fragment>

<Fragment slot="panel.2">

Parameter	Value
Gas price	50 gwei
Callback gas limit	100000
Coordinator gas overhead (Native)	90000
Wrapper gas overhead	13400
Coordinator gas overhead per word	435
Number of random values (words)	2
Wrapper premium percentage	24

1. Calculate the total gas cost:

```

| Gas cost calculation
| Total gas cost |
|
|-----|
|-----|
| Gas price (Coordinator overhead gas + Callback gas limit + Wrapper gas
overhead + (Coordinator overhead gas per word Number of words)) |
|
| 50 gwei x (90000 + 100000 + 13400 + (435 \ 2))
| 10213500 gwei (0.0102135 ETH) |

```

1. Apply the premium percentage to get the total cost of a request:

Applying premium percentage	Request cost
(ETH)	

Total gas cost (ETH) \ ((100 + premium percentage)/100)	
(0.0102135 ETH \ (100 + 24))/100	0.01266474 ETH

This example request would cost 0.01266474 ETH.

</Fragment>

</TabsContent>

getting-started.mdx:

```

---
section: vrf
date: Last Modified
title: "Getting Started with Chainlink VRF V2.5"
excerpt: "Using Chainlink VRF"
whatsnext:
{
  "Security Considerations": "/vrf/v2-5/security",
  "Best Practices": "/vrf/v2-5/best-practices",
  "Supported Networks": "/vrf/v2-5/supported-networks",

```

```

}
metadata:
  title: "Getting Started with Chainlink VRF V2.5"
  description: "Learn how to use randomness in your smart contracts using
Chainlink VRF."
  image: "/files/2a242f1-link.png"
---

```

```
import { Aside, CodeSample, ClickToZoom } from "@components"
```

```

<Aside type="note" title="Requirements">
  This guide assumes that you have basic knowledge about writing and deploying
smart contracts. If you are new to smart
  contract development, learn how to Deploy Your First Smart Contract before
  you begin.
</Aside>

```

In this guide, you will learn about generating randomness on blockchains. This includes learning how to implement a Request and Receive cycle with Chainlink oracles and how to consume random numbers with Chainlink VRF in smart contracts.

How is randomness generated on blockchains? What is Chainlink VRF?

Randomness is very difficult to generate on blockchains. This is because every node on the blockchain must come to the same conclusion and form a consensus. Even though random numbers are versatile and useful in a variety of blockchain applications, they cannot be generated natively in smart contracts. The solution to this issue is Chainlink VRF, also known as Chainlink Verifiable Random Function.

What is the Request and Receive cycle?

The Data Feeds Getting Started guide explains how to consume Chainlink Data Feeds, which consist of reference data posted onchain by oracles. This data is stored in a contract and can be referenced by consumers until the oracle updates the data again.

Randomness, on the other hand, cannot be reference data. If the result of randomness is stored onchain, any actor could retrieve the value and predict the outcome. Instead, randomness must be requested from an oracle, which generates a number and a cryptographic proof. Then, the oracle returns that result to the contract that requested it. This sequence is known as the Request and Receive cycle.

What is the payment process for generating a random number?

VRF requests receive funding from subscription accounts. The Subscription Manager lets you create an account and pre-pay for VRF requests, so that funding of all your application requests are managed in a single location. To learn more about VRF requests funding, see Subscription limits.

How can I use Chainlink VRF?

In this section, you will create an application that uses Chainlink VRF to generate randomness. The contract used in this application has a Game of Thrones theme.

After the contract requests randomness from Chainlink VRF, the result of the randomness will transform into a number between 1 and 20, mimicking the rolling of a 20 sided die. Each number represents a Game of Thrones house. If the dice land on the value 1, the user is assigned house Targaryan, 2 for Lannister, and so on. A full list of houses can be found [here](#).

When rolling the dice, it uses an address variable to track which address is

assigned to each house.

The contract has the following functions:

- rollDice: This submits a randomness request to Chainlink VRF
- fulfillRandomWords: The function that the Oracle uses to send the result back
- house: To see the assigned house of an address

Note: to jump straight to the entire implementation, you can open the VRFD20.sol contract in Remix.

Create and fund a subscription

Chainlink VRF requests receive funding from subscription accounts. The Subscription Manager lets you create an account and pre-pay your use of Chainlink VRF requests.

For this example, create a new subscription on the Sepolia testnet as explained [here](#).

Your subscription has two balances - one for LINK and one for the native token you're using (in this case, Sepolia ETH). You can choose to pay for VRF requests using either balance.

Importing contracts

Chainlink maintains a library of contracts that make consuming data from oracles easier. For Chainlink VRF, you will use:

- VRFConsumerBaseV2Plus that must be imported and extended from the contract that you create.
- VRFV2PlusClient to format your requests to VRF.

```
{/ prettier-ignore /}  
solidity  
// SPDX-License-Identifier: MIT  
pragma solidity 0.8.19;  
  
import {VRFConsumerBaseV2Plus} from  
"@chainlink/contracts/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";  
import {VRFV2PlusClient} from  
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";  
  
contract VRFD20 is VRFConsumerBaseV2Plus {  
  
}
```

Contract variables

This example is adapted for Sepolia testnet but you can change the configuration and make it run for any supported network.

```
{/ prettier-ignore /}  
solidity  
uint256 ssubscriptionId;  
address vrfCoordinator = 0x9DdfaCa8183c41ad55329BdeeD9F6A8d53168B1B;  
bytes32 skeyHash =  
0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae;  
uint32 callbackGasLimit = 40000;  
uint16 requestConfirmations = 3;  
uint32 numWords = 1;
```

- uint256 ssubscriptionId: The subscription ID that this contract uses for

funding requests. Initialized in the constructor.

- address vrfCoordinator: The address of the Chainlink VRF Coordinator contract.
- bytes32 skeyHash: The gas lane key hash value, which is the maximum gas price you are willing to pay for a request in wei. It functions as an ID of the offchain VRF job that runs in response to requests.
- uint32 callbackGasLimit: The limit for how much gas to use for the callback request to your contract's fulfillRandomWords function. It must be less than the maxGasLimit on the coordinator contract. Adjust this value for larger requests depending on how your fulfillRandomWords function processes and stores the received random values. If your callbackGasLimit is not sufficient, the callback will fail and your subscription is still charged for the work done to generate your requested random values.
- uint16 requestConfirmations: How many confirmations the Chainlink node should wait before responding. The longer the node waits, the more secure the random value is. It must be greater than the minimumRequestBlockConfirmations value on the coordinator contract. For Sepolia, the minimumRequestBlockConfirmations value is 3. You can check this and the other configuration values on the coordinator contract by querying the sconfig value in the Sepolia Etherscan block explorer.
- uint32 numWords: How many random values to request. If you can use several random values in a single callback, you can reduce the amount of gas that you spend per random value. In this example, each transaction requests one random value.

To keep track of addresses that roll the dice, the contract uses mappings. Mappings are unique key-value pair data structures similar to hash tables in Java.

```
{/ prettier-ignore /}  
solidity  
mapping(uint256 => address) private srollers;  
mapping(address => uint256) private sresults;
```

- srollers stores a mapping between the requestId (returned when a request is made), and the address of the roller. This is so the contract can keep track of who to assign the result to when it comes back.
- sresults stores the roller and the result of the dice roll.

Initializing the contract

The subscription ID must be initialized in the constructor of the contract. To use VRFConsumerBaseV2Plus properly, you must also pass the VRF coordinator address into its constructor.

The address that creates the smart contract is the owner of the contract.

```
{/ prettier-ignore /}  
solidity  
// SPDX-License-Identifier: MIT  
pragma solidity 0.8.19;  
  
import {VRFConsumerBaseV2Plus} from  
"@chainlink/contracts/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";  
import {VRFV2PlusClient} from  
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";  
  
contract VRFD20 is VRFConsumerBaseV2Plus {  
    // variables  
    // ...  
  
    // constructor  
    constructor(uint256 subscriptionId) VRFConsumerBaseV2Plus(vrfCoordinator) {  
        ssubscriptionId = subscriptionId;  
    }  
}
```

```
    ...  
}
```

rollDice function

The rollDice function will complete the following tasks:

1. Check if the roller has already rolled since each roller can only ever be assigned to a single house.
1. Request randomness by calling the VRF coordinator.
1. Store the requestId and roller address.
1. Emit an event to signal that the dice is rolling.

You must add a ROLLINPROGRESS constant to signify that the dice has been rolled but the result is not yet returned. Also add a DiceRolled event to the contract.

Only the owner of the contract can execute the rollDice function.

This rollDice function is configured so that you pay for VRF requests using LINK by default. If you want to pay for your VRF request with Sepolia ETH instead, set nativePayment to true.

```
{/ prettier-ignore /}  
solidity  
// SPDX-License-Identifier: MIT  
pragma solidity 0.8.19;  
  
import {VRFConsumerBaseV2Plus} from  
"@chainlink/contracts/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";  
import {VRFV2PlusClient} from  
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";  
  
contract VRFD20 is VRFConsumerBaseV2Plus {  
    // variables  
    uint256 private constant ROLLINPROGRESS = 42;  
    // ...  
  
    // events  
    event DiceRolled(uint256 indexed requestId, address indexed roller);  
    // ...  
  
    // ...  
    // { constructor }  
    // ...  
  
    // rollDice function  
    function rollDice(address roller) public onlyOwner returns (uint256  
requestId) {  
        require(sresults[roller] == 0, "Already rolled");  
        // Will revert if subscription is not set and funded.  
  
        requestId = svrfCoordinator.requestRandomWords(  
            VRFV2PlusClient.RandomWordsRequest({  
                keyHash: skeyHash,  
                subId: ssubscriptionId,  
                requestConfirmations: requestConfirmations,  
                callbackGasLimit: callbackGasLimit,  
                numWords: numWords,  
                // Set nativePayment to true to pay for VRF requests with  
                // Sepolia ETH instead of LINK  
                extraArgs:  
                VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: false}))  
            })  
        );  
    }  
}
```



```

    );

    srollers[requestId] = roller;
    sresults[roller] = ROLLINPROGRESS;
    emit DiceRolled(requestId, roller);
  }
}

```

fulfillRandomWords function

fulfillRandomWords is a special function defined within the VRFConsumerBaseV2Plus contract that our contract extends from. The coordinator sends the result of our generated randomWords back to fulfillRandomWords. You will implement some functionality here to deal with the result:

1. Change the result to a number between 1 and 20 inclusively. Note that randomWords is an array that could contain several random values. In this example, request 1 random value.
1. Assign the transformed value to the address in the sresults mapping variable.
1. Emit a DiceLanded event.

```

{/ prettier-ignore /}
solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import {VRFConsumerBaseV2Plus} from
"@chainlink/contracts/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";
import {VRFV2PlusClient} from
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";

contract VRFD20 is VRFConsumerBaseV2Plus {
    // ...
    // { variables }
    // ...

    // events
    // ...
    event DiceLanded(uint256 indexed requestId, uint256 indexed result);

    // ...
    // { constructor }
    // ...

    // ...
    // { rollDice function }
    // ...

    // fulfillRandomWords function
    function fulfillRandomWords(uint256 requestId, uint256[] calldata
randomWords) internal override {

        // transform the result to a number between 1 and 20 inclusively
        uint256 d20Value = (randomWords[0] % 20) + 1;

        // assign the transformed value to the address in the sresults mapping
variable
        sresults[srollers[requestId]] = d20Value;

        // emitting event to signal that dice landed
        emit DiceLanded(requestId, d20Value);
    }
}

```

house function

Finally, the house function returns the house of an address.

To have a list of the house's names, create the getHouseName function that is called in the house function.

```
{/ prettier-ignore /}
solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import {VRFConsumerBaseV2Plus} from
"@chainlink/contracts/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";
import {VRFV2PlusClient} from
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";

contract VRFD20 is VRFConsumerBaseV2Plus {
    // ...
    // { variables }
    // ...

    // ...
    // { events }
    // ...

    // ...
    // { constructor }
    // ...

    // ...
    // { rollDice function }
    // ...

    // ...
    // { fulfillRandomWords function }
    // ...

    // house function
    function house(address player) public view returns (string memory) {
        // dice has not yet been rolled to this address
        require(sresults[player] != 0, "Dice not rolled");

        // not waiting for the result of a thrown dice
        require(sresults[player] != ROLLINPROGRESS, "Roll in progress");

        // returns the house name from the name list function
        return getHouseName(sresults[player]);
    }

    // getHouseName function
    function getHouseName(uint256 id) private pure returns (string memory) {
        // array storing the list of house's names
        string[20] memory houseNames = [
            "Targaryen",
            "Lannister",
            "Stark",
            "Tyrell",
            "Baratheon",
            "Martell",
            "Tully",
            "Bolton",
```

```

        "Greyjoy",
        "Arryn",
        "Frey",
        "Mormont",
        "Tarley",
        "Dayne",
        "Umber",
        "Valeryon",
        "Manderly",
        "Clegane",
        "Glover",
        "Karstark"
    ];

    // returns the house name given an index
    return houseNames[id - 1];
}
}

```

```

{/ prettier-ignore /}
<CodeSample src="samples/VRF/v2-5/VRFD20.sol" showButtonOnly/>

```

You have now completed all necessary functions to generate randomness and assign the user a Game of Thrones house. We've added a few helper functions in there to make using the contract easier and more flexible. You can deploy and interact with the complete contract in Remix.

How do I deploy to testnet?

You will deploy this contract on the Sepolia test network. You must have some Sepolia testnet ETH in your MetaMask account to pay for the gas for each contract interaction with the Sepolia network.

You can choose to pay for your VRF requests in either Sepolia ETH or testnet LINK. The rollDice function is written to use LINK by default. If you want to change to using Sepolia, set nativePayment to true before deploying your contract.

You can request both testnet LINK and testnet ETH from faucets.chain.link/sepolia. Testnet ETH is also available from several public faucets.

This deployment is slightly different than the example in the Deploy Your First Contract guide. In this case, you pass in parameters to the constructor upon deployment.

Once compiled, you'll see a dropdown menu that looks like this in the deploy pane:

```

<ClickToZoom src="/images/vrf/v2-5/select-vrfd20-contract.png" alt="Remix
contract selected" />

```

Select the VRFD20 contract or the name that you gave to your contract.

Click the caret arrow on the right hand side of Deploy to expand the parameter fields, and paste your subscription ID.

```

<ClickToZoom src="/images/vrf/v2-5/deploy-with-sub-id.png" alt="Remix contract
parameters to deploy" />

```

Then click the Deploy button and use your MetaMask account to confirm the transaction.

<Aside type="note" title="Address, Key Hashes and more">

For a full reference of the addresses, key hashes and fees for each network, see VRF Supported Networks.

</Aside>

At this point, your contract should be successfully deployed. However, it can't request anything because it is not yet approved to use the LINK or Sepolia ETH balance in your subscription. If you click rollDice, the transaction will revert.

How do I add my contract to my subscription account?

After you deploy your contract, you must add it as an approved consumer contract so it can use the subscription balance when requesting for randomness.

1. Find your contract address in Remix under Deployed Contracts on the bottom left, and copy the contract address:

<ClickToZoom src="/images/vrf/v2-5/copy-contract-address.png" alt="Copy Remix contract address" />

1. Go to the Subscription Manager, open the details page for your subscription, and add your deployed contract address to the list of consumers:

<ClickToZoom src="/images/vrf/v2-5/ui-add-consumer.png" alt="Add a consumer to your VRF subscription" />

How do I test rollDice?

After you open the deployed contract tab in the bottom left, the function buttons are available. Find rollDice and click the caret to expand the parameter fields. Enter an Ethereum address to specify a "dice roller", and click rollDice.

It takes a few minutes for the transaction to confirm and the response to be sent back. You can get your house by clicking the house function button with the address passed in rollDice. After the response is sent back, you'll be assigned a Game of Thrones house!

Further reading

To read more about generating random numbers in Solidity, read our blog posts:

- 35+ Blockchain RNG Use Cases Enabled by Chainlink VRF
- How to Build Dynamic NFTs on Polygon
- Scaling Onchain Verifiable Randomness With Chainlink VRF v2.5

migration-from-v1.mdx:

section: vrf

date: Last Modified

title: "Migrating from VRF v1"

import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"

import { Aside, CodeSample } from "@components"

import { Tabs, TabsContent } from "@components/Tabs"

<Vrf25Common callout="security" />

VRF V2.5 replaces both VRF V1 and VRF V2 on November 29, 2024. Learn more about

VRF V2.5.

Comparing VRF v1 to VRF v2.5

Chainlink VRF v2.5 includes several improvements and changes to the way you fund and request randomness for your smart contracts:

- You have the option to manage payment for your VRF requests by pre-funding a subscription account, or to directly fund your consuming contracts as you do with VRF V1. Compare subscription and direct funding.
- VRF v2.5 introduces the option to pay for requests in either LINK or native tokens. This choice is available for both subscription and direct funding.

New billing options

- **Native billing:** You have the option to use either native tokens or LINK to pay for VRF requests. Instead of a flat LINK fee per request, there is percentage-based premium fee applied to each request. See the [Billing](#) page for more details. To find out the premium percentages for the networks you use, see the [Supported Networks](#) page.

- **Subscription management:** Chainlink VRF v2.5 has a Subscription Manager application that allows smart contract applications to pre-fund multiple requests for randomness using one subscription account. This reduces the gas fees for VRF requests by eliminating the need to transfer funds for each individual request. You transfer funds to the subscription balance only when it requires additional funding.

- **Unified Billing - Delegate Subscription Balance to Multiple Addresses:** Chainlink VRF v2.5 allows up to 100 smart contract addresses to fund their requests for verifiable randomness from a single subscription account, which is managed by the subscription owner.

- **Variable Callback Gas Limit:** Chainlink VRF v2.5 lets you adjust the callback gas limit when your smart contract application receives verifiable randomness. Consuming contracts can execute more complex logic in the callback request function that receives the random values. Tasks involving the delivered randomness are handled during the response process. The new gas limits are higher than the VRF V1 limit, and vary depending on the underlying blockchain you use. See the gas limits on the [VRF Supported Networks](#) page.

- **More configuration capability:** You can define how many block confirmations must pass before verifiable randomness is generated and delivered onchain when your application makes a request transaction. The range is from 3 to 200 blocks. VRF V1 always waited 10 blocks on Ethereum before delivering onchain randomness. Select a value that protects your application from block re-organizations while still providing sufficiently low latency from request to response. See the [Security Considerations](#) page to learn more.

- **Multiple Random Outputs in a Single Request:** In VRF v2.5, you can request multiple random numbers (multi-word) in a single onchain transaction, which reduces gas costs. The fulfillment is also a single transaction, which reduces the latency of responses.

For direct funding, the configurations for overhead gas have changed:

- The amount of wrapper overhead gas is reduced compared to V2.
- The amount of coordinator overhead gas used varies depending on the network used for your request, whether you're paying in LINK or native tokens, and how many random values you want in each VRF request. See the [Billing](#) page for more details and examples. The new configurations are listed in the [Supported Networks](#) page.

Updating your applications to use VRF v2.5

You have the option to manage payment for your VRF requests with a subscription account, or to directly fund your consuming contracts as you do with VRF V1. Compare subscription and direct funding.

```
{/ prettier-ignore /}  
<TabsContent sharedStore="vrfMethod" client:visible>  
<Fragment slot="tab.1">Subscription</Fragment>  
<Fragment slot="tab.2">Direct funding</Fragment>  
<Fragment slot="panel.1">
```

To modify your existing smart contract code to work with VRF v2.5, complete the following changes. See the [Get a Random Number](#) guide for an example.

1. Set up and fund a subscription in the Subscription Manager at vrf.chain.link.
{/ prettier-ignore /}

```
<div class="remix-callout">  
  <a href="https://vrf.chain.link">Open the Subscription Manager</a>  
</div>
```

1. Add the following imports to your contract:

- VRFConsumerBaseV2Plus. Remove the v1 VRFConsumerBase.sol import. VRFConsumerBaseV2Plus includes the fulfillRandomWords function. The VRFConsumerBaseV2Plus contract imports the IVRFCoordinatorV2Plus interface, which includes the requestRandomWords function.
- VRFV2PlusClient is a library used to format your VRF requests.

```
solidity  
import { VRFConsumerBaseV2Plus } from  
"@chainlink/contracts/src/v0.8/vrf/dev/VRFConsumerBaseV2Plus.sol";  
import { VRFV2PlusClient } from  
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";
```

1. Add a VRFConsumerBaseV2Plus constructor, passing in the LINK token address for the network you're using, as shown in the [Get a Random Number](#) example.

1. Change requestRandomness function calls to requestRandomWords. The requestRandomWords function requires several additional parameters. The extraArgs key allows you to add extra arguments related to new VRF features. Use the nativePayment argument to enable or disable payment in native tokens.

```
{/ prettier-ignore /}  
solidity  
uint256 requestId = svrfCoordinator.requestRandomWords(  
  VRFV2PlusClient.RandomWordsRequest({  
    keyHash: keyHash,  
    subId: svrfSubscriptionId,  
    requestConfirmations: requestConfirmations,  
    callbackGasLimit: callbackGasLimit,  
    numWords: numWords,  
    extraArgs:  
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: true}))  
  })  
);
```

1. Change fulfillRandomness function calls to fulfillRandomWords. Update the call to handle the returned uint256[] array instead of the single uint256 variable.

1. Use the setCoordinator function in your contract so that you can easily

update the VRF coordinator for future VRF releases. This function is inherited from the IVRFCoordinatorV2Plus interface.

```
</Fragment>
<Fragment slot="panel.2">
```

To modify your existing smart contract code to work with VRF v2.5, complete the following changes. See the [Get a Random Number](#) guide for an example.

1. Import and inherit the VRFV2PlusWrapperConsumerBase contract and remove the v1 VRFConsumerBase.sol import. This contract includes the fulfillRandomWords function.

1. Add a VRFV2PlusWrapperConsumerBase constructor, passing in the VRF wrapper address for the network you're using, as shown in the [Get a Random Number](#) example.

1. You can still call the requestRandomness function. However, the v2 requestRandomness function requires several different parameters. See the [Supported networks](#) page to adjust them for your own needs.

- The requestRandomness function in the wrapper contract requires a new extraArgs argument that allows you to add extra arguments related to new VRF features. Use the nativePayment argument to enable or disable payment in native tokens.

- Additionally, the requestRandomness function now returns two arguments instead of one: the request ID and the request price.

```
{/ prettier-ignore /}
solidity
bytes memory extraArgs = VRFV2PlusClient.argsToBytes(
    VRFV2PlusClient.ExtraArgsV1({nativePayment: false})
);
(uint256 reqId, uint256 reqPrice) = requestRandomness(
    callbackGasLimit,
    requestConfirmations,
    numWords,
    extraArgs
);
```

1. If you're paying for requests with LINK, you can still call the requestRandomness function. However, if you're paying with native tokens, call the requestRandomnessPayInNative function instead.

- Both functions return two arguments instead of one: the request ID and the request price.

- Both functions require one additional parameter, extraArgs. Use nativePayment to specify whether or not you want to pay for VRF requests using native tokens:

```
{/ prettier-ignore /}
<Tabs sharedStore="feePaymentType" client:visible>
<Fragment slot="tab.1">LINK</Fragment>
<Fragment slot="tab.2">Native tokens</Fragment>
<Fragment slot="panel.1">
{/ prettier-ignore /}
solidity
bytes memory extraArgs = VRFV2PlusClient.argsToBytes(
    VRFV2PlusClient.ExtraArgsV1({nativePayment: false})
);
(uint256 reqId, uint256 reqPrice) = requestRandomness(
    callbackGasLimit,
    requestConfirmations,
```

```

        numWords,
        extraArgs
    );

</Fragment>
<Fragment slot="panel.2">
    {/ prettier-ignore /}
    solidity
    bytes memory extraArgs = VRFV2PlusClient.argsToBytes(
        VRFV2PlusClient.ExtraArgsV1({nativePayment: true}))
    );
    (uint256 reqId, uint256 reqPrice) = requestRandomnessPayInNative(
        callbackGasLimit,
        requestConfirmations,
        numWords,
        extraArgs
    );

</Fragment>

</Tabs>

```

1. Change fulfillRandomness function calls to fulfillRandomWords. Update the call to handle the returned uint256[] array instead of the single uint256 variable.

```

</Fragment>
</TabsContent>

```

Migration walkthrough

VRF v2.5 currently supports subscriptions and direct funding on all supported networks. To migrate, you need to update your existing smart contract code and redeploy your contracts.

If using subscriptions, create and fund a new VRF v2.5 subscription.

For direct funding, deploy the DirectFundingConsumer example:

```
<CodeSample src="samples/VRF/v2-5/DirectFundingConsumer.sol" showButtonOnly />
```

Update your code

To modify your existing smart contract code to work with VRF v2.5, complete the following changes:

```

{/ prettier-ignore /}
<TabsContent sharedStore="vrfMethod" client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
1. Import the VRFConsumerBaseV2Plus contract and remove the v2 VRFConsumerBaseV2
import.

```

1. Import the VRF v2.5 coordinator, VRFCoordinatorV25, and update any old references to the VRF V2 coordinator in your contract.

1. Add a VRFConsumerBaseV2Plus constructor, passing in the LINK token address for the network you're using.

1. Update your requestRandomWords function calls to reflect the new request structure for VRF v2.5. Make sure to include the new extraArgs part of the VRFV2PlusClient.RandomWordsRequest object, and specify whether or not you want to pay for VRF requests using native tokens:


```

    {/ prettier-ignore /}
    <Tabs sharedStore="feePaymentType" client:visible>
    <Fragment slot="tab.1">LINK</Fragment>
    <Fragment slot="tab.2">Native tokens</Fragment>
    <Fragment slot="panel.1">

    {/ prettier-ignore /}
    solidity
    uint256 requestId = svrfCoordinator.requestRandomWords(
        VRFV2PlusClient.RandomWordsRequest({
            keyHash: keyHash,
            subId: svrfSubscriptionId,
            requestConfirmations: requestConfirmations,
            callbackGasLimit: callbackGasLimit,
            numWords: numWords,
            extraArgs:
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: false}))
        })
    );

    </Fragment>
    <Fragment slot="panel.2">

    {/ prettier-ignore /}
    solidity
    uint256 requestId = svrfCoordinator.requestRandomWords(
        VRFV2PlusClient.RandomWordsRequest({
            keyHash: keyHash,
            subId: svrfSubscriptionId,
            requestConfirmations: requestConfirmations,
            callbackGasLimit: callbackGasLimit,
            numWords: numWords,
            extraArgs:
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: true}))
        })
    );

    </Fragment>
    </Tabs>

```

1. When using the @chainlink/contracts package version 1.1.1 and later, update your fulfillRandomWords function signature to match the VRFConsumerBaseV2Plus contract, which has changed to:

```

function fulfillRandomWords(uint256 requestId, uint256[] calldata
randomWords)

```

In the @chainlink/contracts package version 1.1.0 and earlier, the randomWords parameter has a memory storage location.

```

</Fragment>
<Fragment slot="panel.2">

```

1. Import the VRFV2PlusWrapperConsumerBase contract and remove the v2 VRFV2WrapperConsumerBase import.

1. Add a VRFV2PlusWrapperConsumerBase constructor, passing in the VRF wrapper address for the network you're using. Unlike in V2, you don't have to pass the LINK token address to the constructor.

1. If you're paying for requests with LINK, you can still call the `requestRandomness` function. However, if you're paying with native tokens, call the `requestRandomnessPayInNative` function instead.

Both functions require one additional parameter, `extraArgs`. Use `nativePayment` to specify whether or not you want to pay for VRF requests using native tokens:

```
    {/ prettier-ignore /}
    <Tabs sharedStore="feePaymentType" client:visible>
    <Fragment slot="tab.1">LINK</Fragment>
    <Fragment slot="tab.2">Native tokens</Fragment>
    <Fragment slot="panel.1">

    {/ prettier-ignore /}
    solidity
    bytes memory extraArgs = VRFV2PlusClient.argsToBytes(
        VRFV2PlusClient.ExtraArgsV1({nativePayment: false})
    );
    (uint256 reqId, uint256 reqPrice) = requestRandomness(
        callbackGasLimit,
        requestConfirmations,
        numWords,
        extraArgs
    );

    </Fragment>

    <Fragment slot="panel.2">

    {/ prettier-ignore /}
    solidity
    bytes memory extraArgs = VRFV2PlusClient.argsToBytes(
        VRFV2PlusClient.ExtraArgsV1({nativePayment: true})
    );
    (uint256 reqId, uint256 reqPrice) = requestRandomnessPayInNative(
        callbackGasLimit,
        requestConfirmations,
        numWords,
        extraArgs
    );

    </Fragment>

    </Tabs>
```

1. The V2.5 `requestRandomness` and `requestRandomnessPayInNative` functions both return a tuple: (uint256 requestId, uint256 requestPrice). Adjust your `requestRandomWords` function or any other functions in your code where you call the V2.5 wrapper's `requestRandomness` or `requestRandomnessPayInNative` functions.

1. Make sure your contract has a `withdraw` function for both native tokens and LINK. Both are included in the direct funding example code and the `VRFV2PlusWrapperConsumerExample` contract.

```
</Fragment>
</TabsContent>
```

[View example code](#)

[View example code for both VRF 2.5 subscription and direct funding:](#)

Open the full example SubscriptionConsumer contract:

```
<CodeSample src="samples/VRF/v2-5/SubscriptionConsumer.sol" showButtonOnly />
```

Open the full example DirectFundingConsumer contract:

```
<CodeSample src="samples/VRF/v2-5/DirectFundingConsumer.sol" showButtonOnly />
```

```
# migration-from-v2.mdx:
```

```
---
```

```
section: vrf
```

```
date: Last Modified
```

```
title: "Migrating from VRF v2"
```

```
---
```

```
import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"
```

```
import { Aside, CodeSample } from "@components"
```

```
import { Tabs, TabsContent } from "@components/Tabs"
```

```
<Vrf25Common callout="security" />
```

VRF V2.5 replaces both VRF V1 and VRF V2 on November 29, 2024. Learn more about VRF V2.5.

Benefits of VRF v2.5

Chainlink VRF v2.5 includes all the same key benefits as VRF v2, along with the following additional benefits and changes:

- Easier upgrades to future versions by using the new setCoordinator function
- The option to pay for requests in either LINK or native tokens
- New, flexible request format in requestRandomWords to make any future upgrades easier

Code changes

VRF v2.5 introduces a new request format and the setCoordinator function. See the full migration walkthrough or the code example for more details.

New request format

The request format for VRF v2.5 has changed:

```
{/ prettier-ignore /}
```

```
<TabsContent sharedStore="vrfMethod" client:visible>
```

```
<Fragment slot="tab.1">Subscription</Fragment>
```

```
<Fragment slot="tab.2">Direct funding</Fragment>
```

```
<Fragment slot="panel.1">
```

The requestRandomWords function now uses VRFV2PlusClient.RandomWordsRequest with an object labeling each part of the request:

```
{/ prettier-ignore /}
```

```
solidity
```

```
uint256 requestId = svrfCoordinator.requestRandomWords(
```

```
    VRFV2PlusClient.RandomWordsRequest({
```

```
        keyHash: keyHash,
```

```
        subId: svrfSubscriptionId,
```

```
        requestConfirmations: requestConfirmations,
```

```
        callbackGasLimit: callbackGasLimit,
```

```
        numWords: numWords,
```

```
        extraArgs:
```

```
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: true}))
```

```
    })  
};
```

You must include a value for the new `extraArgs` key, which allows you to add extra arguments related to new VRF features. Use the `nativePayment` argument to enable or disable payment in native tokens.

```
</Fragment>
```

```
<Fragment slot="panel.2">
```

The `requestRandomness` function in the wrapper contract requires a new `extraArgs` argument that allows you to add extra arguments related to new VRF features. Use the `nativePayment` argument to enable or disable payment in native tokens.

Additionally, the `requestRandomness` function now returns two arguments instead of one: the request ID and the request price.

```
{/ prettier-ignore /}  
solidity  
bytes memory extraArgs = VRFV2PlusClient.argsToBytes(  
    VRFV2PlusClient.ExtraArgsV1({nativePayment: false})  
);  
(uint256 reqId, uint256 reqPrice) = requestRandomness(  
    callbackGasLimit,  
    requestConfirmations,  
    numWords,  
    extraArgs  
);
```

```
</Fragment>
```

```
</TabsContent>
```

setCoordinator function

Add the `setCoordinator` function to your contract so that you can easily update the VRF coordinator for future VRF releases.

Subscription ID type change

Note that the subscription ID has changed types from `uint64` in VRF V2 to `uint256` in VRF V2.5.

Billing changes

You have the option to use either native tokens or LINK to pay for VRF requests. To accommodate this, the premium fee has changed from a flat LINK premium amount per request, to a percentage-based premium per request. Refer to the [Billing](#) page for more details. To find out the new premium percentages for the networks you use, see the [Supported Networks](#) page.

For direct funding, the configurations for overhead gas have changed:

- The amount of wrapper overhead gas is reduced compared to V2.
- The amount of coordinator overhead gas used varies depending on the network used for your request, whether you're paying in LINK or native tokens, and how many random values you want in each VRF request. Refer to the [Billing](#) page for more details and examples, and see the new configurations on the [Supported Networks](#) page.

Migration walkthrough

VRF v2.5 currently supports subscriptions and direct funding on all supported networks. To migrate, you need to update your existing smart contract code and

redeploy your contracts.

If using subscriptions, create and fund a new VRF v2.5 subscription.

For direct funding, deploy the DirectFundingConsumer example:

```
<CodeSample src="samples/VRF/v2-5/DirectFundingConsumer.sol" showButtonOnly />
```

Update your code

To modify your existing smart contract code to work with VRF v2.5, complete the following changes:

```
{/ prettier-ignore /}  
<TabsContent sharedStore="vrfMethod" client:visible>  
<Fragment slot="tab.1">Subscription</Fragment>  
<Fragment slot="tab.2">Direct funding</Fragment>  
<Fragment slot="panel.1">  
1. Import the VRFConsumerBaseV2Plus contract and remove the v2 VRFConsumerBaseV2  
import.
```

1. Import the VRF v2.5 coordinator, VRFCoordinatorV25, and update any old references to the VRF V2 coordinator in your contract.

1. Add a VRFConsumerBaseV2Plus constructor, passing in the LINK token address for the network you're using.

1. Update your requestRandomWords function calls to reflect the new request structure for VRF v2.5. Make sure to include the new extraArgs part of the VRFV2PlusClient.RandomWordsRequest object, and specify whether or not you want to pay for VRF requests using native tokens:

```
{/ prettier-ignore /}  
<Tabs sharedStore="feePaymentType" client:visible>  
<Fragment slot="tab.1">LINK</Fragment>  
<Fragment slot="tab.2">Native tokens</Fragment>  
<Fragment slot="panel.1">  
  
{/ prettier-ignore /}  
solidity  
uint256 requestId = svrfCoordinator.requestRandomWords(  
    VRFV2PlusClient.RandomWordsRequest({  
        keyHash: keyHash,  
        subId: svrfSubscriptionId,  
        requestConfirmations: requestConfirmations,  
        callbackGasLimit: callbackGasLimit,  
        numWords: numWords,  
        extraArgs:  
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: false}))  
    })  
);  
  
</Fragment>  
<Fragment slot="panel.2">  
  
{/ prettier-ignore /}  
solidity  
uint256 requestId = svrfCoordinator.requestRandomWords(  
    VRFV2PlusClient.RandomWordsRequest({  
        keyHash: keyHash,  
        subId: svrfSubscriptionId,  
        requestConfirmations: requestConfirmations,  
        callbackGasLimit: callbackGasLimit,
```

```

        numWords: numWords,
        extraArgs:
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment: true}))
    })
    );
</Fragment>
</Tabs>

```

1. When using the @chainlink/contracts package version 1.1.1 and later, update your fulfillRandomWords function signature to match the VRFConsumerBaseV2Plus contract, which has changed to:

```

function fulfillRandomWords(uint256 requestId, uint256[] calldata
randomWords)

```

In the @chainlink/contracts package version 1.1.0 and earlier, the randomWords parameter has a memory storage location.

```

</Fragment>
<Fragment slot="panel.2">

```

1. Import the VRFV2PlusWrapperConsumerBase contract and remove the v2 VRFV2WrapperConsumerBase import.

1. Add a VRFV2PlusWrapperConsumerBase constructor, passing in the VRF wrapper address for the network you're using. Unlike in V2, you don't have to pass the LINK token address to the constructor.

1. If you're paying for requests with LINK, you can still call the requestRandomness function. However, if you're paying with native tokens, call the requestRandomnessPayInNative function instead.

Both functions require one additional parameter, extraArgs. Use nativePayment to specify whether or not you want to pay for VRF requests using native tokens:

```

    {/ prettier-ignore /}
    <Tabs sharedStore="feePaymentType" client:visible>
    <Fragment slot="tab.1">LINK</Fragment>
    <Fragment slot="tab.2">Native tokens</Fragment>
    <Fragment slot="panel.1">

    {/ prettier-ignore /}
    solidity
    bytes memory extraArgs = VRFV2PlusClient.argsToBytes(
        VRFV2PlusClient.ExtraArgsV1({nativePayment: false})
    );
    (uint256 reqId, uint256 reqPrice) = requestRandomness(
        callbackGasLimit,
        requestConfirmations,
        numWords,
        extraArgs
    );

    </Fragment>

    <Fragment slot="panel.2">

    {/ prettier-ignore /}
    solidity

```

```

        bytes memory extraArgs = VRFV2PlusClient.argsToBytes(
            VRFV2PlusClient.ExtraArgsV1({nativePayment: true})
        );
        (uint256 reqId, uint256 reqPrice) = requestRandomnessPayInNative(
            callbackGasLimit,
            requestConfirmations,
            numWords,
            extraArgs
        );
    }
}

```

</Fragment>

</Tabs>

1. The V2.5 requestRandomness and requestRandomnessPayInNative functions both return a tuple: (uint256 requestId, uint256 requestPrice). Adjust your requestRandomWords function or any other functions in your code where you call the V2.5 wrapper's requestRandomness or requestRandomnessPayInNative functions.

1. Make sure your contract has a withdraw function for both native tokens and LINK. Both are included in the direct funding example code and the VRFV2PlusWrapperConsumerExample contract.

</Fragment>

</TabsContent>

Compare example code

Subscription example code

The example SubscriptionConsumer contract shows the migration steps above, applied to the example code from this VRF V2 tutorial. Both of these examples use the subscription method.

Open the full example SubscriptionConsumer contract:

<CodeSample src="samples/VRF/v2-5/SubscriptionConsumer.sol" showButtonOnly />

Compare the major changes between V2.5 and V2:

```

{/ prettier-ignore /}
<TabsContent sharedStore="vrfVersion" client:visible>
<Fragment slot="tab.1">VRF V2.5 example code</Fragment>
<Fragment slot="tab.2">VRF V2 example code</Fragment>
<Fragment slot="panel.1">

```

```

{/ prettier-ignore /}
solidity
// SPDX-License-Identifier: MIT
// An example of a consumer contract that relies on a subscription for funding.
pragma solidity 0.8.19;

```

```

///// UPDATE IMPORTS TO V2.5 /////
import {VRFCustomerBaseV2Plus} from
"@chainlink/contracts/src/v0.8/vrf/dev/VRFCustomerBaseV2Plus.sol";
import {VRFV2PlusClient} from
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";

```

...

/\\

- THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY.

- THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE.
- DO NOT USE THIS CODE IN PRODUCTION.
\\

```
///// INHERIT NEW CONSUMER BASE CONTRACT /////
contract SubscriptionConsumer is VRFConsumerBaseV2Plus {
    ...
    ///// No need to declare a coordinator variable /////
    ///// Use the svrfCoordinator from VRFConsumerBaseV2Plus.sol /////

    ///// SUBSCRIPTION ID IS NOW UINT256 /////
    uint256 ssubscriptionId;

    ...

    ///// USE NEW KEYHASH FOR VRF 2.5 GAS LANE /////
    // For a list of available gas lanes on each network,
    // see https://docs.chain.link/docs/vrf/v2-5/supported-networks
    bytes32 keyHash =
        0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae;

    ...

    ///// USE NEW CONSUMER BASE CONSTRUCTOR /////
    constructor(
        ///// UPDATE TO UINT256 /////
        uint256 subscriptionId
    )
        VRFConsumerBaseV2Plus(0x9DdfaCa8183c41ad55329BdeeD9F6A8d53168B1B)
    {
        ssubscriptionId = subscriptionId;
    }

    function requestRandomWords()
        external
        onlyOwner
        returns (uint256 requestId)
    {
        ///// UPDATE TO NEW V2.5 REQUEST FORMAT /////
        // To enable payment in native tokens, set nativePayment to true.
        // Use the svrfCoordinator from VRFConsumerBaseV2Plus.sol
        requestId = svrfCoordinator.requestRandomWords(
            VRFV2PlusClient.RandomWordsRequest({
                keyHash: keyHash,
                subId: ssubscriptionId,
                requestConfirmations: requestConfirmations,
                callbackGasLimit: callbackGasLimit,
                numWords: numWords,
                extraArgs: VRFV2PlusClient.argsToBytes(
                    VRFV2PlusClient.ExtraArgsV1({nativePayment: false})
                )
            })
        );
        ...
    }
    ...
}

}
```

</Fragment>

<Fragment slot="panel.2">

{/ prettier-ignore /}


```

solidity
// SPDX-License-Identifier: MIT
// An example of a consumer contract that relies on a subscription for funding.
pragma solidity ^0.8.7;

///// USES V2 IMPORTS /////
import {VRFCoordinatorV2Interface} from
"@chainlink/contracts/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol";
import {VRFConsumerBaseV2} from
"@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";
import {ConfirmedOwner} from
"@chainlink/contracts/src/v0.8/shared/access/ConfirmedOwner.sol";

/\

- THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY.
- THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE.
- DO NOT USE THIS CODE IN PRODUCTION.
  \

///// USES V2 CONSUMER BASE CONTRACT /////
contract VRFv2Consumer is VRFConsumerBaseV2, ConfirmedOwner {
    ...

    ///// OLD TYPE FOR SUBSCRIPTION ID /////
    uint64 ssubscriptionId;

    ...

    ///// KEYHASH FOR VRF V2 GAS LANE /////
    // The gas lane to use, which specifies the maximum gas price to bump to.
    // For a list of available gas lanes on each network,
    // see https://docs.chain.link/docs/vrf/v2/subscription/supported-networks/
    #configurations
    bytes32 keyHash =
        0x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c;

    ...

    ///// USES V2 CONSUMER BASE AND COORDINATOR CONSTRUCTORS /////
    constructor(
        uint64 subscriptionId
    )
        VRFConsumerBaseV2(0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625)
        ConfirmedOwner(msg.sender)
    {
        COORDINATOR = VRFCoordinatorV2Interface(
            0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625
        );
        ssubscriptionId = subscriptionId;
    }

    ...

    function requestRandomWords()
        external
        onlyOwner
        returns (uint256 requestId)
    {
        ///// USES V2 REQUEST FORMAT /////
        requestId = COORDINATOR.requestRandomWords(
            keyHash,
            ssubscriptionId,
            requestConfirmations,
            callbackGasLimit,

```

```

        numWords
    );
    ...
}

...
}

```

```

</Fragment>
</TabsContent>

```

Direct funding example code

The example DirectFundingConsumer contract shows the migration steps above, applied to the example code from this VRF V2 tutorial. Both of these examples use the direct funding method.

Open the full example DirectFundingConsumer contract:

```
<CodeSample src="samples/VRF/v2-5/DirectFundingConsumer.sol" showButtonOnly />
```

Compare the major changes between V2.5 and V2:

```

{/ prettier-ignore /}
<TabsContent sharedStore="vrfVersion" client:visible>
<Fragment slot="tab.1">VRF V2.5 example code</Fragment>
<Fragment slot="tab.2">VRF V2 example code</Fragment>
<Fragment slot="panel.1">

{/ prettier-ignore /}
solidity
// SPDX-License-Identifier: MIT
// An example of a consumer contract that directly pays for each request.
pragma solidity 0.8.20;

///// UPDATE IMPORTS TO V2.5 /////
import {ConfirmedOwner} from
"@chainlink/contracts/src/v0.8/shared/access/ConfirmedOwner.sol";
import {VRFV2PlusWrapperConsumerBase} from
"@chainlink/contracts/src/v0.8/vrf/dev/VRFV2PlusWrapperConsumerBase.sol";
import {LinkTokenInterface} from
"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.sol";
import {VRFV2PlusClient} from
"@chainlink/contracts/src/v0.8/vrf/dev/libraries/VRFV2PlusClient.sol";

/
THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY.
THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE.
DO NOT USE THIS CODE IN PRODUCTION.
/

///// INHERIT NEW WRAPPER CONSUMER BASE CONTRACT /////
contract DirectFundingConsumer is VRFV2PlusWrapperConsumerBase, ConfirmedOwner {

    ...
    ///// USE NEW WRAPPER CONSUMER BASE CONSTRUCTOR /////
    constructor()
        ConfirmedOwner(msg.sender)
        VRFV2PlusWrapperConsumerBase(wrapperAddress) ///// ONLY PASS IN WRAPPER
ADDRESS /////
    {}

    function requestRandomWords(

```

```

        bool enableNativePayment
    ) external onlyOwner returns (uint256) {
        ////// UPDATE TO NEW V2.5 REQUEST FORMAT: ADD EXTRA ARGS ////
        bytes memory extraArgs = VRFV2PlusClient.argsToBytes(
            VRFV2PlusClient.ExtraArgsV1({nativePayment: enableNativePayment})
        );
        uint256 requestId;
        uint256 reqPrice;
        if (enableNativePayment) {
            ////// USE THIS FUNCTION TO PAY IN NATIVE TOKENS ////
            (requestId, reqPrice) = requestRandomnessPayInNative(
                callbackGasLimit,
                requestConfirmations,
                numWords,
                extraArgs ////// PASS IN EXTRA ARGS ////
            );
        } else {
            ////// USE THIS FUNCTION TO PAY IN LINK ////
            (requestId, reqPrice) = requestRandomness(
                callbackGasLimit,
                requestConfirmations,
                numWords,
                extraArgs ////// PASS IN EXTRA ARGS ////
            );
        }
        ...
        return requestId;
    }
    ...
}

```

</Fragment>

<Fragment slot="panel.2">

```

{/ prettier-ignore /}
solidity
// SPDX-License-Identifier: MIT
// An example of a consumer contract that directly pays for each request.
pragma solidity ^0.8.7;

```

```

////// USES V2 IMPORTS ////
import {ConfirmedOwner} from
"@chainlink/contracts/src/v0.8/shared/access/ConfirmedOwner.sol";
import {VRFV2WrapperConsumerBase} from
"@chainlink/contracts/src/v0.8/vrf/VRFV2WrapperConsumerBase.sol";
import {LinkTokenInterface} from
"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.sol";

```

/\

```

- THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY.
- THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE.
- DO NOT USE THIS CODE IN PRODUCTION.
  \

```

```

////// USES V2 WRAPPER CONSUMER BASE CONTRACT ////
contract VRFv2DirectFundingConsumer is
VRFV2WrapperConsumerBase,
ConfirmedOwner
{
    ...

```

```

////// USES V2 WRAPPER CONSUMER BASE CONSTRUCTOR ////

```

```

    constructor()
        ConfirmedOwner(msg.sender)
        VRFV2WrapperConsumerBase(linkAddress, wrapperAddress) ///// TWO
PARAMETERS /////
    {}

    function requestRandomWords()
        external
        onlyOwner
        returns (uint256 requestId)
    {
        ///// USES V2 REQUEST FORMAT /////
        requestId = requestRandomness(
            callbackGasLimit,
            requestConfirmations,
            numWords
        );
        ...
        return requestId;
    }

    ...
}

```

</Fragment>
</TabsContent>

security.mdx:

```

---
section: vrf
date: Last Modified
title: "VRF Security Considerations"
---

```

Gaining access to high quality randomness onchain requires a solution like Chainlink's VRF, but it also requires you to understand some of the ways that miners or validators can potentially manipulate randomness generation. Here are some of the top security considerations you should review in your project.

- Use requestId to match randomness requests with their fulfillment in order
- Choose a safe block confirmation time, which will vary between blockchains
- Do not allow re-requesting or cancellation of randomness
- Don't accept bids/bets/inputs after you have made a randomness request
- The fulfillRandomWords function must not revert
- Use VRFConsumerBaseV2Plus in your contract to interact with the VRF service

Use requestId to match randomness requests with their fulfillment in order

If your contract could have multiple VRF requests in flight simultaneously, you must ensure that the order in which the VRF fulfillments arrive cannot be used to manipulate your contract's user-significant behavior.

Blockchain miners/validators can control the order in which your requests appear onchain, and hence the order in which your contract responds to them.

For example, if you made randomness requests A, B, C in short succession, there is no guarantee that the associated randomness fulfillments will also be in order A, B, C. The randomness fulfillments might just as well arrive at your contract in order C, A, B or any other order.

We recommend using the requestID to match randomness requests with their corresponding fulfillments.

Choose a safe block confirmation time, which will vary between blockchains

In principle, miners/validators of your underlying blockchain could rewrite the chain's history to put a randomness request from your contract into a different block, which would result in a different VRF output. Note that this does not enable a miner to determine the random value in advance. It only enables them to get a fresh random value that might or might not be to their advantage. By way of analogy, they can only re-roll the dice, not predetermine or predict which side it will land on.

You must choose an appropriate confirmation time for the randomness requests you make. Confirmation time is how many blocks the VRF service waits before writing a fulfillment to the chain to make potential rewrite attacks unprofitable in the context of your application and its value-at-risk.

Do not allow re-requesting or cancellation of randomness

Any re-request or cancellation of randomness is an incorrect use of VRF v2.5. dApps that implement the ability to cancel or re-request randomness for specific commitments must consider the additional attack vectors created by this capability. For example, you must prevent the ability for any party to discard unfavorable randomness.

Don't accept bids/bets/inputs after you have made a randomness request

Consider the example of a contract that mints a random NFT in response to a user's actions.

The contract should:

1. Record whatever actions of the user may affect the generated NFT.
1. Stop accepting further user actions that might affect the generated NFT and issue a randomness request.
1. On randomness fulfillment, mint the NFT.

Generally speaking, whenever an outcome in your contract depends on some user-supplied inputs and randomness, the contract should not accept any additional user-supplied inputs after it submits the randomness request.

Otherwise, the cryptoeconomic security properties may be violated by an attacker that can rewrite the chain.

fulfillRandomWords must not revert

If your fulfillRandomWords() implementation reverts, the VRF service will not attempt to call it a second time. Make sure your contract logic does not revert. Consider simply storing the randomness and taking more complex follow-on actions in separate contract calls made by you, your users, or an Automation Node.

Use VRFConsumerBaseV2Plus in your contract, to interact with the VRF service

If you implement the subscription method, use VRFConsumerBaseV2Plus. It includes a check to ensure the randomness is fulfilled by VRFCoordinatorV25. For this reason, it is a best practice to inherit from VRFConsumerBaseV2Plus. Similarly, don't override rawFulfillRandomness.

```
# supported-networks.mdx:
```

```
---
```

```
section: vrf
```

```

date: Last Modified
title: "Supported Networks"
metadata:
  title: "Chainlink VRF v2.5 Supported Networks"
  linkToWallet: true
  image: "/files/OpenGraphV3.png"
---

```

```

import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"
import ResourcesCallout from
"@features/resources/callouts/ResourcesCallout.astro"
import { Address, Aside, CopyText } from "@components"
import { TabsContent } from "@components/Tabs"

```

```
<Vrf25Common callout="security" />
```

Chainlink VRF allows you to integrate provably fair and verifiably random data in your smart contract.

For implementation details, read [Introduction to Chainlink VRF](#).

Coordinator parameters

These parameters are configured in the coordinator contract. You can view these values by running `getConfig` on the coordinator or by viewing the coordinator contracts in a blockchain explorer.

- `uint16 minimumRequestConfirmations`: The minimum number of confirmation blocks on VRF requests before oracles respond
- `uint32 maxGasLimit`: The maximum gas limit supported for a `fulfillRandomWords` callback.
- `uint32 stalenessSeconds`: How long the coordinator waits until we consider the ETH/LINK price used for converting gas costs to LINK is stale and use `fallbackWeiPerUnitLink`
- `uint32 gasAfterPaymentCalculation`: How much gas is used outside of the payment calculation. This covers the additional operations required to decrement the subscription balance and increment the balance for the oracle that handled the request.

Configurations

VRF v2.5 coordinators for subscription funding are available on several networks.

```
<ResourcesCallout callout="bridgeRisks" />
```

Ethereum mainnet

```

{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
| Item | Value |
| ----- |
|
----- |
| LINK Token | <Address
contractUrl="https://etherscan.io/token/0x514910771AF9Ca656af840dff83E8264EcF986
CA" urlId="10x514910771AF9Ca656af840dff83E8264EcF986CA" urlClass="erc-token-
address"/> |
| VRF Coordinator | <Address

```

```

contractUrl="https://etherscan.io/address/0xD7f86b4b8Cae7D942340FF628F82735b7a20
893a" />
|
| 200 gwei Key Hash      | <CopyText
text="0x8077df514608a09f83e4e8d300645594e5d7234665448ba83f51a50f842bd3d9" code/>
|
| 500 gwei Key Hash      | <CopyText
text="0x3fd2fec10d06ee8f65e7f2e95f5c56511359ece3f33960ad8a866ae24a8ff10b" code/>
|
| 1000 gwei Key Hash     | <CopyText
text="0xc6bf2e7b88e5cfbb4946ff23af846494ae1f3c65270b79ee7876c9aa99d3d45f" code/>
|
| Premium percentage <br/> (paying with ETH)      | 24
| Premium percentage <br/> (paying with LINK)      | 20
| Max Gas Limit          | 2,500,000
| Minimum Confirmations  | 3
| Maximum Confirmations  | 200
| Maximum Random Values  | 500
|
</Fragment>
<Fragment slot="panel.2">
| Item                    | Value
| -----
| -----
| -----
| LINK Token              | <Address
contractUrl="https://etherscan.io/token/0x514910771AF9Ca656af840dff83E8264EcF986
CA" urlId="10x514910771AF9Ca656af840dff83E8264EcF986CA" urlClass="erc-token-
address"/> |
| VRF Wrapper             | <Address
contractUrl="https://etherscan.io/address/0x02aae1A04f9828517b3007f83f6181900CaD
910c" /> |
| VRF Coordinator         | <Address
contractUrl="https://etherscan.io/address/0xD7f86b4b8Cae7D942340FF628F82735b7a20
893a" /> |
| 200 gwei Key Hash      | <CopyText
text="0x8077df514608a09f83e4e8d300645594e5d7234665448ba83f51a50f842bd3d9" code/>
|
| 500 gwei Key Hash      | <CopyText
text="0x3fd2fec10d06ee8f65e7f2e95f5c56511359ece3f33960ad8a866ae24a8ff10b" code/>
|
| 1000 gwei Key Hash     | <CopyText
text="0xc6bf2e7b88e5cfbb4946ff23af846494ae1f3c65270b79ee7876c9aa99d3d45f" code/>
|
| Premium percentage <br/> (paying with ETH) | 24
| Premium percentage <br/> (paying with LINK) | 20
| Minimum Confirmations          | 3
| Maximum Confirmations          | 200
| Maximum Random Values          | 10
| Wrapper Gas overhead           | 13400

```

```
| Coordinator Gas Overhead (Native) | 90000
| Coordinator Gas Overhead (LINK) | 112000
| Coordinator Gas Overhead per Word | 435
|
```

```
</Fragment>
</TabsContent>
```

Sepolia testnet

```
<Aside type="note" title="Sepolia Faucets">
  Testnet LINK and ETH are available from <a
href="https://faucets.chain.link/sepolia">faucets.chain.link</a>.<br />
  Testnet ETH is also available from several public <a
href="https://faucetlink.to/sepolia">ETH faucets</a>.
</Aside>
```

```
{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
```

```
| Item | Value
| ----- |
```

```
-----
----- |
| LINK Token | <Address
contractUrl="https://sepolia.etherscan.io/token/0x779877A7B0D9E8603169DdbD7836e4
78b4624789" urlId="111551110x779877A7B0D9E8603169DdbD7836e478b4624789"
urlClass="erc-token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://sepolia.etherscan.io/address/0x9DdfaCa8183c41ad55329BdeeD9F
6A8d53168B1B" />
```

```
|
| 100 gwei Key Hash | <CopyText
text="0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae" code/>
```

```
| Premium percentage <br/> (paying with Sepolia ETH) | 24
```

```
| Premium percentage <br/> (paying with testnet LINK) | 20
```

```
| Max Gas Limit | 2,500,000
```

```
| Minimum Confirmations | 3
```

```
| Maximum Confirmations | 200
```

```
| Maximum Random Values | 500
```

```
</Fragment>
<Fragment slot="panel.2">
```

```
| Item | Value
| ----- |
```

```
-----
----- |
| LINK Token | <Address
contractUrl="https://sepolia.etherscan.io/token/0x779877A7B0D9E8603169DdbD7836e4
78b4624789" urlId="111551110x779877A7B0D9E8603169DdbD7836e478b4624789"
```



```

urlClass="erc-token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://sepolia.etherscan.io/address/0x195f15F2d49d693cE265b4fB0fdD
bE15b1850Cc1" />
|
| VRF Coordinator | <Address
contractUrl="https://sepolia.etherscan.io/address/0x9DdfaCa8183c41ad55329BdeeD9F
6A8d53168B1B" />
|
| 100 gwei Key Hash | <CopyText
text="0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae" code/>
|
| Premium percentage <br/> (paying with Sepolia ETH) | 24
| Premium percentage <br/> (paying with testnet LINK) | 20
|
| Minimum Confirmations | 3
| Maximum Confirmations | 200
| Maximum Random Values | 10
| Wrapper Gas overhead | 13400
| Coordinator Gas Overhead (Native) | 90000
| Coordinator Gas Overhead (LINK) | 112000
| Coordinator Gas Overhead per Word | 435
|
</Fragment>
</TabsContent>

```

BNB Chain

<Aside type="tip" title="Important">

The LINK provided by the BNB Chain Bridge is not ERC-677 compatible, so cannot be used with Chainlink oracles. However, it can be converted to the official LINK token on BNB Chain using Chainlink's PegSwap service.

</Aside>

```
{/ prettier-ignore /}
```

```
<TabsContent client:visible>
```

```
<Fragment slot="tab.1">Subscription</Fragment>
```

```
<Fragment slot="tab.2">Direct funding</Fragment>
```

```
<Fragment slot="panel.1">
```

```
| Item | Value
```

```
| ----- |
```

```
-----
```

```
----- |
```

```
| LINK Token | <Address
```

```
contractUrl="https://bscscan.com/token/0x404460C6A5EdE2D891e8297795264fDe62ADBB7
5" urlId="560x404460C6A5EdE2D891e8297795264fDe62ADBB75" urlClass="erc-token-
address"/> |
```

```
| VRF Coordinator | <Address
```

```
contractUrl="https://bscscan.com/address/0xd691f04bc0C9a24Edb78af9E005Cf85768F69
4C9" />
```

```
| 200 gwei Key Hash | <CopyText
```

```
text="0x130dba50ad435d4ecc214aad0d5820474137bd68e7e77724144f27c3c377d3d4" code/>
```

```
|
```

```

| 500 gwei Key Hash      | <CopyText
text="0xeb0f72532fed5c94b4caf7b49caf454b35a729608a441101b9269efb7efe2c6c" code/>
|
| 1000 gwei Key Hash     | <CopyText
text="0xb94a4fdb12830e15846df59b27d7c5d92c9c24c10cf6ae49655681ba560848dd" code/>
|
| Premium percentage <br/> (paying with BNB)      | 60
| Premium percentage <br/> (paying with LINK)     | 50
| Max Gas Limit          | 2,500,000
| Minimum Confirmations  | 3
| Maximum Confirmations  | 200
| Maximum Random Values  | 500
|
</Fragment>
<Fragment slot="panel.2">
| Item                      | Value
| ----- |
|
| LINK Token                | <Address
contractUrl="https://bscscan.com/token/0x404460C6A5EdE2D891e8297795264fDe62ADBB7
5" urlId="560x404460C6A5EdE2D891e8297795264fDe62ADBB75" urlClass="erc-token-
address"/> |
| VRF Wrapper               | <Address
contractUrl="https://bscscan.com/address/0x471506e6ADED0b9811D05B8cAc8Db25eE839A
c94" />
|
| VRF Coordinator          | <Address
contractUrl="https://bscscan.com/address/0xd691f04bc0C9a24Edb78af9E005Cf85768F69
4C9" />
|
| 200 gwei Key Hash        | <CopyText
text="0x130dba50ad435d4ecc214aad0d5820474137bd68e7e77724144f27c3c377d3d4" code/>
|
| 500 gwei Key Hash        | <CopyText
text="0xeb0f72532fed5c94b4caf7b49caf454b35a729608a441101b9269efb7efe2c6c" code/>
|
| 1000 gwei Key Hash       | <CopyText
text="0xb94a4fdb12830e15846df59b27d7c5d92c9c24c10cf6ae49655681ba560848dd" code/>
|
| Premium percentage <br/> (paying with testnet BNB) | 60
| Premium percentage <br/> (paying with testnet LINK) | 50
| Minimum Confirmations    | 3
| Maximum Confirmations    | 200
| Maximum Random Values    | 10
| Wrapper Gas overhead     | 13400
| Coordinator Gas Overhead (Native) | 99500
| Coordinator Gas Overhead (LINK)   | 121500
| Coordinator Gas Overhead per Word | 435

```

```
|
</Fragment>
</TabsContent>
```

BNB Chain testnet

```
{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
<Aside type="note" title="BNB Chain Faucet">
  Testnet LINK is available from https://faucets.chain.link/bnb-chain-testnet
</Aside>
```

Item	Value
LINK Token	https://testnet.bscscan.com/address/0x84b9B910527Ad5C03A9Ca831909E21e236EA7b06
VRF Coordinator	https://testnet.bscscan.com/address/0xDA3b641D438362C440Ac5458c57e00a712b66700
50 gwei Key Hash	0x8596b430971ac45bdf6088665b9ad8e8630c9d5049ab54b14dff711bee7c0e26
Premium percentage (paying with testnet BNB)	60
Premium percentage (paying with testnet LINK)	50
Max Gas Limit	2,500,000
Minimum Confirmations	3
Maximum Confirmations	200
Maximum Random Values	500

```
</Fragment>
<Fragment slot="panel.2">
  Item      Value
  LINK Token  <Address
contractUrl="https://testnet.bscscan.com/address/0x84b9B910527Ad5C03A9Ca831909E21e236EA7b06" urlId="970x84b9B910527Ad5C03A9Ca831909E21e236EA7b06" urlClass="erc-token-address"/>
  VRF Wrapper  <Address
contractUrl="https://testnet.bscscan.com/address/0x471506e6ADED0b9811D05B8cAc8Db25eE839Ac94" />
  VRF Coordinator  <Address
contractUrl="https://testnet.bscscan.com/address/0xDA3b641D438362C440Ac5458c57e00a712b66700" />
```

```
| 50 gwei Key Hash | <CopyText  
text="0x8596b430971ac45bdf6088665b9ad8e8630c9d5049ab54b14dff711bee7c0e26"  
code />
```

```
| Premium percentage <br/> (paying with testnet BNB) | 60
```

```
| Premium percentage <br/> (paying with testnet LINK) | 50
```

```
| Minimum Confirmations | 3
```

```
| Maximum Confirmations | 200
```

```
| Maximum Random Values | 10
```

```
| Wrapper Gas overhead | 13400
```

```
| Coordinator Gas Overhead (Native) | 99500
```

```
| Coordinator Gas Overhead (LINK) | 121500
```

```
| Coordinator Gas Overhead per Word | 435
```

```
</Fragment>
```

```
</TabsContent>
```

Polygon mainnet

```
<Aside type="tip" title="Important">
```

The LINK provided by the Polygon Bridge is not ERC-677 compatible,
so cannot be used with Chainlink oracles. However, it can be converted to the
official LINK token on Polygon using
Chainlink's PegSwap service

```
</Aside>
```

```
{/ prettier-ignore /}
```

```
<TabsContent client:visible>
```

```
<Fragment slot="tab.1">Subscription</Fragment>
```

```
<Fragment slot="tab.2">Direct funding</Fragment>
```

```
<Fragment slot="panel.1">
```

```
| Item | Value
```

```
| ----- |
```

```
-----
```

```
----- |
```

```
| LINK Token | <Address
```

```
contractUrl="https://polygonscan.com/address/0xb0897686c545045aFc77CF20eC7A532E3  
120E0F1" urlId="1370xb0897686c545045aFc77CF20eC7A532E3120E0F1" urlClass="erc-  
token-address"/> |
```

```
| VRF Coordinator | <Address
```

```
contractUrl="https://polygonscan.com/address/0xec0Ed46f36576541C75739E915ADbCb3D  
E24bd77" />
```

```
| 200 gwei Key Hash | <CopyText  
text="0x0ffb9bd0c1c18c0263dd778dadd1d64240d7bc338d95fec1cf0473928ca7eaf9e" code/>
```

```
| 500 gwei Key Hash | <CopyText  
text="0x719ed7d7664abc3001c18aac8130a2265e1e70b7e036ae20f3ca8b92b3154d86"  
code />
```

```
| 1000 gwei Key Hash | <CopyText  
text="0x192234a5cda4cc07c0b66dfbcfb785341cc790edc50032e842667dbb506cada"  
code />
```

```

| Premium percentage <br/> (paying with POL) | 84
| Premium percentage <br/> (paying with LINK) | 70
| Max Gas Limit | 2,500,000
| Minimum Confirmations | 3
| Maximum Confirmations | 200
| Maximum Random Values | 500
</Fragment>
<Fragment slot="panel.2">
| Item | Value
| ----- |
-----
----- |
| LINK Token | <Address
contractUrl="https://polygonscan.com/address/0xb0897686c545045aFc77CF20eC7A532E3
120E0F1" urlId="1370xb0897686c545045aFc77CF20eC7A532E3120E0F1" urlClass="erc-
token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://polygonscan.com/address/0xc8F13422c49909F4Ec24BF65EDFBEbe41
0BB9D7c" />
|
| VRF Coordinator | <Address
contractUrl="https://polygonscan.com/address/0xec0Ed46f36576541C75739E915ADbCb3D
E24bD77" />
|
| 200 gwei Key Hash | <CopyText
text="0x0ffb0c1c18c0263dd778dadd1d64240d7bc338d95fec1cf0473928ca7eaf9e" code/>
|
| 500 gwei Key Hash | <CopyText
text="0x719ed7d7664abc3001c18aac8130a2265e1e70b7e036ae20f3ca8b92b3154d86"
code />
|
| 1000 gwei Key Hash | <CopyText
text="0x192234a5cda4cc07c0b66dfbcfb785341cc790edc50032e842667dbb506cada"
code />
|
| Premium percentage <br/> (paying with POL) | 84
| Premium percentage <br/> (paying with LINK) | 70
| Minimum Confirmations | 3
| Maximum Confirmations | 200
| Maximum Random Values | 10
| Wrapper Gas overhead | 13400
| Coordinator Gas Overhead (Native) | 99500
| Coordinator Gas Overhead (LINK) | 121500
| Coordinator Gas Overhead per Word | 435
</Fragment>
</TabsContent>

```

Polygon Amoy testnet

```
{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
| Item | Value
| ----- |
|-----|
|-----|
| LINK Token | <Address
contractUrl="https://amoy.polygonscan.com/address/0x0fd9e8d3af1aabee056eb9e802c3a
762a667b1904" urlId="800020x0fd9e8d3af1aabee056eb9e802c3a762a667b1904"
urlClass="erc-token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://amoy.polygonscan.com/address/0x343300b5d84D444B2ADc9116FEF1
bED02BE49Cf2" />
|
| 500 gwei Key Hash | <CopyText
text="0x816bedba8a50b294e5cbd47842baf240c2385f2eaf719edbd4f250a137a8c899" code/>
|
| Premium percentage <br/> (paying with testnet POL) | 84
| Premium percentage <br/> (paying with testnet LINK) | 70
| Max Gas Limit | 2,500,000
| Minimum Confirmations | 3
| Maximum Confirmations | 200
| Maximum Random Values | 500
</Fragment>
<Fragment slot="panel.2">
| Item | Value
| ----- |
|-----|
|-----|
| LINK Token | <Address
contractUrl="https://amoy.polygonscan.com/address/0x0fd9e8d3af1aabee056eb9e802c3a
762a667b1904" urlId="800020x0fd9e8d3af1aabee056eb9e802c3a762a667b1904"
urlClass="erc-token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://amoy.polygonscan.com/address/0x6e6c366a1cd1F92ba87Fd6f96F74
3B0e6c967Bf0" />
|
| VRF Coordinator | <Address
contractUrl="https://amoy.polygonscan.com/address/0x343300b5d84D444B2ADc9116FEF1
bED02BE49Cf2" />
|
| 500 gwei Key Hash | <CopyText
text="0x816bedba8a50b294e5cbd47842baf240c2385f2eaf719edbd4f250a137a8c899" code/>
|
| Premium percentage <br/> (paying with testnet POL) | 84
| Premium percentage <br/> (paying with testnet LINK) | 70
```

Minimum Confirmations	3
Maximum Confirmations	200
Maximum Random Values	10
Wrapper Gas overhead	13400
Coordinator Gas Overhead (Native)	99500
Coordinator Gas Overhead (LINK)	121500
Coordinator Gas Overhead per Word	435

</Fragment>
</TabsContent>

Avalanche mainnet

```
{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
| Item | Value
| ----- |
|-----|
| LINK Token | <Address
contractUrl="https://snowtrace.io/address/0x5947BB275c521040051D82396192181b4132
27A3" urlId="431140x5947BB275c521040051D82396192181b413227A3" urlClass="erc-
token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://snowtrace.io/address/0xE40895D055bccd2053dD0638C9695E326152
b1A4" />
|
| 200 gwei Key Hash | <CopyText
text="0xea7f56be19583eeb8255aa79f16d8bd8a64cedf68e42fefee1c9ac5372b1a102" code/>
|
| 500 gwei Key Hash | <CopyText
text="0x84213dcadf1f89e4097eb654e3f284d7d5d5bda2bd4748d8b7fada5b3a6eaa0d" code/>
|
| 1000 gwei Key Hash | <CopyText
text="0xe227ebd10a873dde8e58841197a07b410038e405f1180bd117be6f6557fa491c" code/>
|
| Premium percentage <br/> (paying with AVAX) | 60
|
| Premium percentage <br/> (paying with LINK) | 50
|
| Max Gas Limit | 2,500,000
|
| Minimum Confirmations | 0
|
| Maximum Confirmations | 200
|
| Maximum Random Values | 500
|
</Fragment>
<Fragment slot="panel.2">
| Item | Value
|-----|
```

```

| ----- |
-----
----- |
| LINK Token | <Address
contractUrl="https://snowtrace.io/address/0x5947BB275c521040051D82396192181b4132
27A3" urlId="431140x5947BB275c521040051D82396192181b413227A3" urlClass="erc-
token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://snowtrace.io/address/0x62Fb87c10A917580cA99AB9a86E213Eb98aa
820C" />
|
| VRF Coordinator | <Address
contractUrl="https://snowtrace.io/address/0xE40895D055bccd2053dD0638C9695E326152
b1A4" />
|
| 200 gwei Key Hash | <CopyText
text="0xea7f56be19583eeb8255aa79f16d8bd8a64cedf68e42fefee1c9ac5372b1a102" code/>
|
| 500 gwei Key Hash | <CopyText
text="0x84213dcadf1f89e4097eb654e3f284d7d5d5bda2bd4748d8b7fada5b3a6eaa0d" code/>
|
| 1000 gwei Key Hash | <CopyText
text="0xe227ebd10a873dde8e58841197a07b410038e405f1180bd117be6f6557fa491c" code/>
|
| Premium percentage <br/> (paying with AVAX) | 60
| Premium percentage <br/> (paying with LINK) | 50
| Minimum Confirmations | 0
| Maximum Confirmations | 200
| Maximum Random Values | 10
| Wrapper Gas overhead | 13400
| Coordinator Gas Overhead (Native) | 107000
| Coordinator Gas Overhead (LINK) | 129000
| Coordinator Gas Overhead per Word | 435
|
</Fragment>
</TabsContent>

```

Avalanche Fuji testnet

```

<Aside type="note" title="Avax Fuji Faucet">
  Testnet LINK is available from https://faucets.chain.link/fuji
</Aside>

```

```

{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
| Item | Value
| ----- |
-----
----- |
| LINK Token | <Address

```



```

contractUrl="https://testnet.snowtrace.io/address/0x0b9d5D9136855f6FEc3c0993feE6
E9CE8a297846" urlId="431130x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846"
urlClass="erc-token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://testnet.snowtrace.io/address/0x5C210eF41CD1a72de73bF76eC396
37bB0d3d7BEE" />
|
| 300 gwei Key Hash | <CopyText
text="0xc799bd1e3bd4d1a41cd4968997a4e03dfd2a3c7c04b695881138580163f42887" code/>
|
| Premium percentage <br/> (paying with testnet AVAX) | 60
| Premium percentage <br/> (paying with testnet LINK) | 50
| Max Gas Limit | 2,500,000
| Minimum Confirmations | 1
| Maximum Confirmations | 200
| Maximum Random Values | 500
</Fragment>
<Fragment slot="panel.2">
| Item | Value
| ----- |
|-----|
|-----|
| LINK Token | <Address
contractUrl="https://testnet.snowtrace.io/address/0x0b9d5D9136855f6FEc3c0993feE6
E9CE8a297846" urlId="431130x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846"
urlClass="erc-token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://testnet.snowtrace.io/address/0x327B83F409E1D5f13985c6d05844
20FA648f1F56" />
|
| VRF Coordinator | <Address
contractUrl="https://testnet.snowtrace.io/address/0x5C210eF41CD1a72de73bF76eC396
37bB0d3d7BEE" />
|
| 300 gwei Key Hash | <CopyText
text="0xc799bd1e3bd4d1a41cd4968997a4e03dfd2a3c7c04b695881138580163f42887" code/>
|
| Premium percentage <br/> (paying with testnet AVAX) | 60
| Premium percentage <br/> (paying with testnet LINK) | 50
| Minimum Confirmations | 0
| Maximum Confirmations | 200
| Maximum Random Values | 10
| Wrapper Gas overhead | 13400
| Coordinator Gas Overhead (Native) | 104500
| Coordinator Gas Overhead (LINK) | 126500
| Coordinator Gas Overhead per Word | 435

```

```
</Fragment>
</TabsContent>
```

Arbitrum mainnet

```
{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
| Item | Value
| ----- |
|-----|
|-----|
| LINK Token | <Address
contractUrl="https://arbiscan.io/address/0xf97f4df75117a78c1A5a0DBb814Af92458539
FB4" urlId="421610xf97f4df75117a78c1A5a0DBb814Af92458539FB4" urlClass="erc-
token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://arbiscan.io/address/0x3C0Ca683b403E37668AE3DC4FB62F4B29B6f7
a3e" />
|
| 2 gwei Key Hash | <CopyText
text="0x9e9e46732b32662b9adc6f3abdf6c5e926a666d174a4d6b8e39c4cca76a38897" code/>
|
| 30 gwei Key Hash | <CopyText
text="0x8472ba59cf7134dfe321f4d61a430c4857e8b19cdd5230b09952a92671c24409" code/>
|
| 150 gwei Key Hash | <CopyText
text="0xe9f223d7d83ec85c4f78042a4845af3a1c8df7757b4997b815ce4b8d07aca68c" code/>
|
| Premium percentage <br/> (paying with ETH) | 60
| Premium percentage <br/> (paying with LINK) | 50
| Max Gas Limit | 2,500,000
| Minimum Confirmations | 1
| Maximum Confirmations | 200
| Maximum Random Values | 500
|
</Fragment>
<Fragment slot="panel.2">
| Item | Value
| ----- |
|-----|
|-----|
| LINK Token | <Address
contractUrl="https://arbiscan.io/address/0xf97f4df75117a78c1A5a0DBb814Af92458539
FB4" urlId="4216130xf97f4df75117a78c1A5a0DBb814Af92458539FB4" urlClass="erc-
token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://arbiscan.io/address/0x14632CD5c12eC5875D41350B55e825c54406B
aaB" />
|
| VRF Coordinator | <Address
contractUrl="https://arbiscan.io/address/0x3C0Ca683b403E37668AE3DC4FB62F4B29B6f7
```

```

a3e" />
|
| 2 gwei Key Hash          | <CopyText
text="0x9e9e46732b32662b9adc6f3abdf6c5e926a666d174a4d6b8e39c4cca76a38897" code/>
|
| 30 gwei Key Hash        | <CopyText
text="0x8472ba59cf7134dfe321f4d61a430c4857e8b19cdd5230b09952a92671c24409" code/>
|
| 150 gwei Key Hash       | <CopyText
text="0xe9f223d7d83ec85c4f78042a4845af3a1c8df7757b4997b815ce4b8d07aca68c" code/>
|
| Premium percentage <br/> (paying with ETH)  | 60
| Premium percentage <br/> (paying with LINK) | 50
| Minimum Confirmations      | 0
| Maximum Confirmations      | 200
| Maximum Random Values      | 10
| Wrapper Gas overhead       | 13400
| Coordinator Gas Overhead (Native) | 104500
| Coordinator Gas Overhead (LINK)   | 126500
| Coordinator Gas Overhead per Word | 435
|
</Fragment>
</TabsContent>

```

Arbitrum Sepolia testnet

```

<Aside type="note" title="Arbitrum Sepolia Testnet Faucet">
  Testnet ETH and LINK are available from https://faucets.chain.link/arbitrum-sepolia
</Aside>

```

```

{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
| Item          | Value
|
| ----- |
|
| ----- |
| LINK Token    | <Address
contractUrl="https://sepolia.arbiscan.io/address/0xb1D4538B4571d411F07960EF2838Ce337FE1E80E" urlId="4216130xb1D4538B4571d411F07960EF2838Ce337FE1E80E" urlClass="erc-token-address"/> |
|
| VRF Coordinator | <Address
contractUrl="https://sepolia.arbiscan.io/address/0x5CE8D5A2BC84beb22a398CCA51996F7930313D61" />
|
| 50 gwei Key Hash | <CopyText
text="0x1770bdc7eec7771f7ba4ffd640f34260d7f095b79c92d34a5b2551d6f6cfd2be" code/>
|
| Premium percentage <br/> (paying with Sepolia ETH) | 60

```

```

| Premium percentage <br/> (paying with testnet LINK) | 50
| Max Gas Limit | 2,500,000
| Minimum Confirmations | 1
| Maximum Confirmations | 200
| Maximum Random Values | 500
</Fragment>
<Fragment slot="panel.2">
| Item | Value
| ----- |
|-----|
|-----|
| LINK Token | <Address
contractUrl="https://sepolia.arbiscan.io/address/0xb1D4538B4571d411F07960EF2838C
e337FE1E80E" urlId="4216130xb1D4538B4571d411F07960EF2838Ce337FE1E80E"
urlClass="erc-token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://sepolia.arbiscan.io/address/0x29576aB8152A09b9DC634804e4aDE
73dA1f3a3CC" />
|
| VRF Coordinator | <Address
contractUrl="https://sepolia.arbiscan.io/address/0x5CE8D5A2BC84beb22a398CCA51996
F7930313D61" />
|
| 50 gwei Key Hash | <CopyText
text="0x1770bdc7eec7771f7ba4ffd640f34260d7f095b79c92d34a5b2551d6f6cfd2be" code/>
|
| Premium percentage <br/> (paying with Sepolia ETH) | 60
| Premium percentage <br/> (paying with testnet LINK) | 50
| Minimum Confirmations | 0
| Maximum Confirmations | 200
| Maximum Random Values | 10
| Wrapper Gas overhead | 13400
| Coordinator Gas Overhead (Native) | 104500
| Coordinator Gas Overhead (LINK) | 126500
| Coordinator Gas Overhead per Word | 435
</Fragment>
</TabsContent>

BASE mainnet

{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
| Item | Value

```

```

|
| ----- |
|-----|
|-----|
| LINK Token | <Address
contractUrl="https://basescan.org/address/0x88Fb150BDc53A65fe94Dea0c9BA0a6dAf8C6
e196" urlId="84530x88Fb150BDc53A65fe94Dea0c9BA0a6dAf8C6e196" urlClass="erc-
token-address"/> |
| VRF Coordinator | <Address
contractUrl="https://basescan.org/address/0xd5D517aBE5cF79B7e95eC98dB0f0277788aF
F634" />
|
| 2 gwei Key Hash | <CopyText
text="0x00b81b5a830cb0a4009fbd8904de511e28631e62ce5ad231373d3cdad373ccab" code/>
|
| 30 gwei Key Hash | <CopyText
text="0xdc2f87677b01473c763cb0aee938ed3341512f6057324a584e5944e786144d70" code/>
|
| Premium percentage <br/> (paying with BASE) | 60
| Premium percentage <br/> (paying with LINK) | 50
| Max Gas Limit | 2,500,000
| Minimum Confirmations | 0
| Maximum Confirmations | 200
| Maximum Random Values | 500
|
</Fragment>
<Fragment slot="panel.2">
| Item | Value
| ----- |
|-----|
|-----|
| LINK Token | <Address
contractUrl="https://basescan.org/address/0x88Fb150BDc53A65fe94Dea0c9BA0a6dAf8C6
e196" urlId="84530x88Fb150BDc53A65fe94Dea0c9BA0a6dAf8C6e196" urlClass="erc-
token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://basescan.org/address/0xb0407dbe851f8318bd31404A49e658143C98
2F23" />
|
| VRF Coordinator | <Address
contractUrl="https://basescan.org/address/0xd5D517aBE5cF79B7e95eC98dB0f0277788aF
F634" />
|
| 2 gwei Key Hash | <CopyText
text="0x00b81b5a830cb0a4009fbd8904de511e28631e62ce5ad231373d3cdad373ccab" code/>
|
| 30 gwei Key Hash | <CopyText
text="0xdc2f87677b01473c763cb0aee938ed3341512f6057324a584e5944e786144d70" code/>
|
| Premium percentage <br/> (paying with BASE) | 60
| Premium percentage <br/> (paying with LINK) | 50
| Minimum Confirmations | 0

```

```
|
| Maximum Confirmations      | 200
| Maximum Random Values     | 10
| Wrapper Gas overhead      | 13400
| Coordinator Gas Overhead (Native) | 128500
| Coordinator Gas Overhead (LINK)  | 150400
| Coordinator Gas Overhead per Word | 435
|
|</Fragment>
|</TabsContent>
```

BASE Sepolia testnet

```
{/ prettier-ignore /}
<TabsContent client:visible>
<Fragment slot="tab.1">Subscription</Fragment>
<Fragment slot="tab.2">Direct funding</Fragment>
<Fragment slot="panel.1">
| Item                      | Value
| ----- |
|
| ----- |
| LINK Token                | <Address
contractUrl="https://sepolia.basescan.org/address/0xE4aB69C077896252FAFBD49EFD26
B5D171A32410" urlId="845320xE4aB69C077896252FAFBD49EFD26B5D171A32410"
urlClass="erc-token-address"/> |
| VRF Coordinator           | <Address
contractUrl="https://sepolia.basescan.org/address/0x5C210eF41CD1a72de73bF76eC396
37bB0d3d7BEE" /> |
| 30 gwei Key Hash         | <CopyText
text="0x9e1344a1247c8a1785d0a4681a27152bffdb43666ae5bf7d14d24a5efd44bf71" code/>
| Premium percentage <br/> (paying with Sepolia BASE ETH) | 60
| Premium percentage <br/> (paying with LINK) | 50
| Max Gas Limit          | 2,500,000
| Minimum Confirmations | 0
| Maximum Confirmations | 200
| Maximum Random Values | 500
|
|</Fragment>
<Fragment slot="panel.2">
| Item                      | Value
| ----- |
|
| ----- |
| LINK Token                | <Address
contractUrl="https://sepolia.basescan.org/address/0xE4aB69C077896252FAFBD49EFD26
B5D171A32410" urlId="845320xE4aB69C077896252FAFBD49EFD26B5D171A32410"
```

```

urlClass="erc-token-address"/> |
| VRF Wrapper | <Address
contractUrl="https://sepolia.basescan.org/address/0x7a1BaC17Ccc5b313516C5E16fb24
f7659aA5ebed" />
|
| VRF Coordinator | <Address
contractUrl="https://sepolia.basescan.org/address/0x5C210eF41CD1a72de73bF76eC396
37bB0d3d7BEE" />
|
| 30 gwei Key Hash | <CopyText
text="0x9e1344a1247c8a1785d0a4681a27152bffdb4366ae5bf7d14d24a5efd44bf71" code/>
|
| Premium percentage <br/> (paying with Sepolia BASE ETH) | 60
| Premium percentage <br/> (paying with LINK) | 50
| Minimum Confirmations | 0
| Maximum Confirmations | 200
| Maximum Random Values | 10
| Wrapper Gas overhead | 13400
| Coordinator Gas Overhead (Native) | 128500
| Coordinator Gas Overhead (LINK) | 150400
| Coordinator Gas Overhead per Word | 435
</Fragment>
</TabsContent>

```

get-a-random-number.mdx:

```

---
section: vrf
date: Last Modified
title: "Get a Random Number"
whatsnext:
  {
    "Security Considerations": "/vrf/v2-5/security",
    "Best Practices": "/vrf/v2-5/best-practices",
    "Migrating from V2": "/vrf/v2-5/migration-from-v2",
    "Supported Networks": "/vrf/v2-5/supported-networks",
  }
metadata:
  description: "How to generate a random number inside a smart contract using
Chainlink VRF v2.5 - Direct funding method."
---

```

```

import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"
import { Aside, CodeSample } from "@components"
import { TabsContent } from "@components/Tabs"

```

This guide explains how to get random values using a simple contract to request and receive random values from Chainlink VRF v2.5 without managing a subscription. To explore more applications of VRF, refer to our blog.

Requirements

This guide assumes that you know how to create and deploy smart contracts on

Ethereum testnets using the following tools:

- The Remix IDE
- MetaMask
- Sepolia testnet ETH

If you are new to developing smart contracts on Ethereum, see the Getting Started guide to learn the basics.

Create and deploy a VRF compatible contract

For this example, use the `DirectFundingConsumer.sol` sample contract. This contract imports the following dependencies:

- `VRFV2PlusWrapperConsumerBase.sol`([link](#))
- `VRFV2PlusClient.sol`([link](#))

The contract also includes pre-configured values for the necessary request parameters such as `callbackGasLimit`, `requestConfirmations`, the number of random words `numWords`, the VRF v2.5 Wrapper address `wrapperAddress`, and the LINK token address `linkAddress`. You can change these parameters if you want to experiment on different testnets.

Build and deploy the contract on Sepolia.

1. Open the `DirectFundingConsumer.sol` contract in Remix.

```
{/ prettier-ignore /}  
<CodeSample src="samples/VRF/v2-5/DirectFundingConsumer.sol" showButtonOnly/>
```

1. On the Compile tab in Remix, compile the `DirectFundingConsumer` contract.

1. Configure your deployment. On the Deploy tab in Remix, select the Injected Web3 Environment and select the `DirectFundingConsumer` contract from the contract list.

1. Click the Deploy button to deploy your contract onchain. MetaMask opens and asks you to confirm the transaction.

1. After you deploy your contract, copy the address from the Deployed Contracts list in Remix. Before you can request randomness from VRF v2.5, you must fund your consuming contract with enough tokens in order to request for randomness. Next, fund your contract.

Fund your contract

Requests for randomness will fail unless your consuming contract has enough tokens. VRF V2.5 allows you to use either native tokens or LINK to pay for your requests.

1. Acquire testnet LINK and Sepolia ETH.

1. Fund your contract with either testnet LINK or Sepolia ETH, depending on how you want to pay for your VRF requests. For this example, fund your contract with 2 LINK or 0.01 Sepolia ETH. (The actual request cost is closer to 0.877 LINK or 0.001 ETH. You can withdraw the excess funds after you're done with this contract.)

Request random values

The deployed contract requests random values from Chainlink VRF, receives those values, builds a struct `RequestStatus` containing them, and stores the struct in a mapping `srequests`. Run the `requestRandomWords()` function on your contract to start the request.

1. Return to Remix and view your deployed contract functions in the Deployed Contracts list.

1. Expand the requestRandomWords() function. For enableNativePayment, fill in true if you're paying for your request with Sepolia ETH, or false if you're paying with testnet LINK. Click transact to send the request for random values to Chainlink VRF. MetaMask opens and asks you to confirm the transaction.

After you approve the transaction, Chainlink VRF processes your request. Chainlink VRF fulfills the request and returns the random values to your contract in a callback to the fulfillRandomWords() function. At this point, a new key requestId is added to the mapping srequests. Depending on current testnet conditions, it might take a few minutes for the callback to return the requested random values to your contract.

1. To fetch the request ID of your request, call lastRequestId().

1. After the oracle returns the random values to your contract, the mapping srequests is updated. The received random values are stored in srequests[requestId].randomWords.

1. Call getRequestStatus() and specify the requestId to display the random words.

```
<Aside type="note" title="Note on Requesting or Cancelling Randomness">  
  Do not allow re-requesting or cancellation of randomness. For more  
  information, see the VRF Security  
  Considerations page.  
</Aside>
```

Analyzing the contract

In this example, the consuming contract uses static configuration parameters.

```
<CodeSample src="samples/VRF/v2-5/DirectFundingConsumer.sol" />
```

The parameters define how your requests will be processed. You can find the values for your network in the Supported networks page.

- uint32 callbackGasLimit: The limit for how much gas to use for the callback request to your contract's fulfillRandomWords() function. It must be less than the maxGasLimit limit on the coordinator contract minus the wrapperGasOverhead. See the VRF v2.5 Direct funding limits for more details. Adjust this value for larger requests depending on how your fulfillRandomWords() function processes and stores the received random values. If your callbackGasLimit is not sufficient, the callback will fail and your consuming contract is still charged for the work done to generate your requested random values.

- uint16 requestConfirmations: How many confirmations the Chainlink node should wait before responding. The longer the node waits, the more secure the random value is. It must be greater than the minimumRequestBlockConfirmations limit on the coordinator contract.

- uint32 numWords: How many random values to request. If you can use several random values in a single callback, you can reduce the amount of gas that you spend per random value. The total cost of the callback request depends on how your fulfillRandomWords() function processes and stores the received random values, so adjust your callbackGasLimit accordingly.

The contract includes the following functions:

- requestRandomWords(): Takes your specified parameters and submits the request to the VRF v2.5 Wrapper contract.

- fulfillRandomWords(): Receives random values and stores them with your contract.
- getRequestStatus(): Retrieve request details for a given requestId.
- withdrawLink(): At any time, the owner of the contract can withdraw the outstanding LINK balance from it.
- withdrawNative(): At any time, the owner of the contract can withdraw the outstanding native token balance from it.

<Aside type="note" title="Security Considerations">

Be sure to review your contracts to make sure they follow the best practices on the security considerations page.

</Aside>

Clean up

After you are done with this contract, you can retrieve the remaining testnet tokens to use with other examples:

```
{/ prettier-ignore /}
```

```
<TabsContent sharedStore="feePaymentType" client:visible>
```

```
<Fragment slot="tab.1">LINK</Fragment>
```

```
<Fragment slot="tab.2">Native tokens</Fragment>
```

```
<Fragment slot="panel.1">
```

Call the withdrawLink() function. MetaMask opens and asks you to confirm the transaction. After you approve the transaction, the remaining LINK will be transferred from your consuming contract to your wallet address.

```
</Fragment>
```

```
<Fragment slot="panel.2">
```

Call getBalance to retrieve your contract's ETH balance in wei, and then pass that into the withdrawNative() function. MetaMask opens and asks you to confirm the transaction. After you approve the transaction, the remaining Sepolia ETH will be transferred from your consuming contract to your wallet address.

```
</Fragment>
```

```
</TabsContent>
```

direct-funding.mdx:

```
---
```

```
section: vrf
```

```
date: Last Modified
```

```
title: "Direct Funding Method"
```

```
isIndex: true
```

```
whatsnext:
```

```
{
```

```
  "Get a Random Number": "/vrf/v2-5/direct-funding/get-a-random-number",
```

```
  "Supported Networks": "/vrf/v2-5/supported-networks",
```

```
}
```

```
metadata:
```

```
  title: "Generate Random Numbers for Smart Contracts using Chainlink VRF v2.5 - Direct funding method"
```

```
  description: "Learn how to securely generate random numbers for your smart contract with Chainlink VRF v2.5. This guide uses the Direct funding method."
```

```
---
```

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"
```

```
import { Aside, ClickToZoom } from "@components"
```

This guide explains how to generate random numbers using the direct funding method. This method doesn't require a subscription and is optimal for one-off

requests for randomness. This method also works best for applications where your end-users must pay the fees for VRF because the cost of the request is determined at request time.

<Aside type="note" title="Migrate to V2.5">

Follow the migration guide to learn how VRF has changed in V2.5 and to get example code.

</Aside>

Unlike the subscription method, the direct funding method does not require you to create subscriptions and pre-fund them. Instead, you must directly fund consuming contracts with native tokens or LINK before they request randomness. Because the consuming contract directly pays for the request, the cost is calculated during the request and not during the callback when the randomness is fulfilled. Learn how to estimate costs.

Request and receive data

Requests to Chainlink VRF v2.5 follow the request and receive data cycle similarly to VRF V2. This end-to-end diagram shows each step in the lifecycle of a VRF direct funding request:

<ClickToZoom src="/images/vrf/v2-5/direct-funding-architecture-diagram.png" />

Two types of accounts exist in the Ethereum ecosystem, and both are used in VRF:

- EOA (Externally Owned Account): An externally owned account that has a private key and can control a smart contract. Transactions can be initiated only by EOAs.
- Smart contract: A smart contract that does not have a private key and executes what it has been designed for as a decentralized application.

The Chainlink VRF v2.5 solution uses both offchain and onchain components:

- VRF v2.5 Wrapper (onchain component): A wrapper for the VRF Coordinator that provides an interface for consuming contracts.
- VRF v2.5 Coordinator (onchain component): A contract designed to interact with the VRF service. It emits an event when a request for randomness is made, and then verifies the random number and proof of how it was generated by the VRF service.
- VRF service (offchain component): Listens for requests by subscribing to the VRF Coordinator event logs and calculates a random number based on the block hash and nonce. The VRF service then sends a transaction to the VRFCoordinator including the random number and a proof of how it was generated.

Set up your contract and request

Set up your consuming contract:

1. Your contract must inherit VRFV2PlusWrapperConsumerBase.

1. Your contract must implement the fulfillRandomWords function, which is the callback VRF function. Here, you add logic to handle the random values after they are returned to your contract.

1. Submit your VRF request by calling the requestRandomness function in the VRFV2PlusWrapperConsumerBase contract. Include the following parameters in your request:

- requestConfirmations: The number of block confirmations the VRF service will wait to respond. The minimum and maximum confirmations for your network can be found here.
- callbackGasLimit: The maximum amount of gas to pay for completing the

callback VRF function.

- numWords: The number of random numbers to request. You can find the maximum number of random values per request for your network in the Supported networks page.
- extraArgs: A parameter for additional arguments related to new VRF features, such as enabling payment in native tokens.

How VRF processes your request

After you submit your request, it is processed using the Request & Receive Data cycle:

1. The consuming contract calls the VRFV2PlusWrapper calculateRequestPrice function to estimate the total transaction cost to fulfill randomness. Learn how to estimate transaction costs.

1. The consuming contract calls the LinkToken transferAndCall function to pay the wrapper with the calculated request price. This method sends LINK tokens and executes the VRFV2PlusWrapper onTokenTransfer logic.

1. The VRFV2PlusWrapper's onTokenTransfer logic triggers the VRF 2.5 Coordinator requestRandomWords function to request randomness.

1. The VRF coordinator emits an event.

1. The VRF service picks up the event and waits for the specified number of block confirmations to respond back to the VRF coordinator with the random values and a proof (requestConfirmations).

1. The VRF coordinator verifies the proof onchain, then it calls back the wrapper contract's fulfillRandomWords function.

1. Finally, the VRF Wrapper calls back your consuming contract.

Limits

You can see the configuration for each network on the Supported networks page. You can also view the full configuration for each VRF v2.5 Wrapper contract directly in Etherscan. As an example, view the Ethereum Mainnet VRF v2.5 Wrapper contract configuration by calling getConfig function.

- Each wrapper has a maxNumWords parameter that limits the maximum number of random values you can receive in each request.
- The maximum allowed callbackGasLimit value for your requests is defined in the Coordinator contract supported networks page. Because the VRF v2.5 Wrapper adds an overhead, your callbackGasLimit must not exceed maxGasLimit - wrapperGasOverhead. Learn more about estimating costs.

subscription.mdx:

```
---
section: vrf
date: Last Modified
title: "Subscription Method"
isIndex: true
metadata:
  title: "Generate Random Numbers for Smart Contracts using Chainlink VRF v2.5 - Subscription Method"
  description: "Learn how to securely generate random numbers for your smart contract with Chainlink VRF v2.5(an RNG). This guide uses the subscription method."
---
```

```
import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"
import { Aside, ClickToZoom } from "@components"
import { TabsContent } from "@components/Tabs"
import { YouTube } from "@astro-community/astro-embed-youtube"
```

```
<Vrf25Common callout="security" />
```

This section explains how to generate random numbers using the subscription method.

```
<Aside type="note" title="Migrate to V2.5">
```

```
  Follow the migration guide to learn how VRF has changed in V2.5 and to get
  example
  code.
```

```
</Aside>
```

Subscriptions

VRF v2.5 requests receive funding from subscription accounts. Creating a VRF subscription lets you create an account and pre-pay for VRF v2.5, so you don't provide funding each time your application requests randomness. This reduces the total gas cost to use VRF v2.5. It also provides a simple way to fund your use of Chainlink products from a single location, so you don't have to manage multiple wallets across several different systems and applications.

Subscriptions have the following core concepts:

- Subscription id: 256-bit unsigned integer representing the unique identifier of the subscription.
- Subscription accounts: An account that holds LINK and native tokens and makes them available to fund requests to Chainlink VRF v2.5 coordinators.
- Subscription owner: The wallet address that creates and manages a subscription account. Any account can add LINK or native tokens to the subscription balance, but only the owner can add approved consuming contracts or withdraw funds.
- Consumers: Consuming contracts that are approved to use funding from your subscription account.
- Subscription balance: The amount of funds in LINK or native tokens maintained on your subscription account. Your subscription can maintain balances for both LINK and native tokens. Requests from consuming contracts will continue to be funded until the balance runs out, so be sure to maintain sufficient funds in your subscription balance to pay for the requests and keep your applications running.

For Chainlink VRF v2.5 to fulfill your requests, you must maintain a sufficient amount of LINK or native tokens in your subscription balance. Gas cost calculation includes the following variables:

- Gas price: The current gas price, which fluctuates depending on network conditions.
- Callback gas: The amount of gas used for the callback request that returns your requested random values.
- Verification gas: The amount of gas used to verify randomness onchain.

The gas price depends on current network conditions. The callback gas depends on your callback function, and the number of random values in your request. The cost of each request is final only after the transaction is complete, but you define the limits you are willing to spend for the request with the following variables:

- Gas lane: The maximum gas price you are willing to pay for a request in wei. Define this limit by specifying the appropriate keyHash in your request. The limits of each gas lane are important for handling gas price spikes when

Chainlink VRF bumps the gas price to fulfill your request quickly.

- Callback gas limit: Specifies the maximum amount of gas you are willing to spend on the callback request. Define this limit by specifying the callbackGasLimit value in your request.

Request and receive data

Requests to Chainlink VRF v2.5 follow the request and receive data cycle similarly to VRF V2. This end-to-end diagram shows each step in the lifecycle of a VRF subscription request, and registering a smart contract with a VRF subscription account:

<ClickToZoom src="/images/vrf/v2-5/subscription-architecture-diagram.png" />

<Aside type="tip" title="Account types used in VRF">

Two types of accounts exist in the Ethereum ecosystem, and both are used in VRF:

- Externally Owned Account (EOA): An externally owned account that has a private key and can control a smart contract. Transactions can only be initiated by EOAs.
- Smart contract: A contract that does not have a private key and executes what it has been designed for as a decentralized application.

</Aside>

VRF v2.5 uses both offchain and onchain components:

- VRF v2.5 Coordinator (onchain component): A contract designed to interact with the VRF service. It emits an event when a request for randomness is made, and then verifies the random number and proof of how it was generated by the VRF service.
- VRF service (offchain component): Listens for requests by subscribing to the VRF Coordinator event logs and calculates a random number based on the block hash and nonce. The VRF service then sends a transaction to the VRFCoordinator including the random number and a proof of how it was generated.

Set up your contract and request

Set up your consuming contract:

1. Your contract must inherit VRFCustomerBaseV2Plus.

1. Your contract must implement the fulfillRandomWords function, which is the callback VRF function. Here, you add logic to handle the random values after they are returned to your contract.

1. Submit your VRF request by calling requestRandomWords of the VRF Coordinator. Include the following parameters in your request:

- keyHash: Identifier that maps to a job and a private key on the VRF service and that represents a specified gas lane. If your request is urgent, specify a gas lane with a higher gas price limit. The configuration for your network can be found [here](#).
- ssubscriptionId: The subscription ID that the consuming contract is registered to. LINK funds are deducted from this subscription.
- requestConfirmations: The number of block confirmations the VRF service will wait to respond. The minimum and maximum confirmations for your network can be found [here](#).
- callbackGasLimit: The maximum amount of gas a user is willing to pay for completing the callback VRF function. Note that you cannot put a value larger than maxGasLimit of the VRF Coordinator contract (read coordinator contract limits for more details).
- numWords: The number of random numbers to request. The maximum random

values that can be requested for your network can be found [here](#).

In VRF 2.5, the request format has changed:

```
{/ prettier-ignore /}
solidity
uint256 requestId =
svrfCoordinator.requestRandomWords(VRFV2PlusClient.RandomWordsRequest({
    keyHash: keyHash,
    subId: subId,
    requestConfirmations: requestConfirmations,
    callbackGasLimit: callbackGasLimit,
    numWords: numWords,
    extraArgs:
VRFV2PlusClient.argsToBytes(VRFV2PlusClient.ExtraArgsV1({nativePayment:
true}))) // new parameter
}))
);
```

1. Add the setCoordinator method to your contract. This makes it easier to update your contract for future VRF releases by setting the new coordinator.

How VRF processes your request

After you submit your request, it is processed using the Request & Receive Data cycle. The VRF coordinator processes the request and determines the final charge to your subscription using the following steps:

1. The VRF coordinator emits an event.

1. The VRF service picks up the event and waits for the specified number of block confirmations to respond back to the VRF coordinator with the random values and a proof (requestConfirmations).

1. The VRF coordinator verifies the proof onchain, then it calls back the consuming contract fulfillRandomWords function.

Limits

Chainlink VRF v2.5 has subscription limits and coordinator contract limits.

Subscription limits

Subscriptions are required to maintain a minimum balance, and they can support a limited number of consuming contracts.

Minimum subscription balance

Each subscription must maintain a minimum balance to fund requests from consuming contracts. This minimum balance requirement serves as a buffer against gas volatility by ensuring that all your requests have more than enough funding to go through. If your balance is below the minimum, your requests remain pending for up to 24 hours before they expire. After you add sufficient LINK or native tokens to a subscription, pending requests automatically process as long as they have not expired.

In the Subscription Manager, the minimum subscription balance is displayed as the Max Cost, and it indicates the amount of LINK or native tokens you need to add for a pending request to process. After the request is processed, only the amount actually consumed by the request is deducted from your balance. For example, if you are paying for your VRF requests in LINK and your minimum balance is 10 LINK, but your subscription balance is 5 LINK, you need to add at least 5 more LINK for your request to process. This does not mean that your

request will ultimately cost 10 LINK. If the request ultimately costs 3 LINK after it has processed, then 3 LINK is deducted from your subscription balance. The same concept applies if you are paying in native tokens.

The minimum subscription balance must be sufficient for each new consuming contract that you add to a subscription. For example, the minimum balance for a subscription that supports 20 consuming contracts needs to cover all the requests for all 20 contracts, while a subscription with one consuming contract only needs to cover that one contract.

For one request, the required size of the minimum balance depends on the gas lane and the size of the request. For example, a consuming contract that requests one random value will require a smaller minimum balance than a consuming contract that requests 50 random values. In general, you can estimate the required minimum balance using the following formula where max verification gas is always 200,000 gwei.

The following formulas show how the minimum subscription balance is calculated for LINK and native tokens in general. Specific examples of each are available on the Billing page, where you can compare the higher minimum subscription balance with the lower amount for an actual request.

```
{/ prettier-ignore /}  
<TabsContent sharedStore="feePaymentMethod" client:visible>  
<Fragment slot="tab.1">LINK</Fragment>  
<Fragment slot="tab.2">Native tokens</Fragment>  
<Fragment slot="panel.1">
```

```
((Gas lane maximum (Max verification gas + Callback gas limit)) (100 +  
premium %)/100) / (1,000,000,000 Gwei/ETH)) / (ETH/LINK price) = Minimum LINK
```

Here is the same formula, broken out into steps:

```
Gas lane maximum (Max verification gas + Callback gas limit) = Total estimated  
gas (Gwei)  
Total estimated gas (Gwei) ((100 + premium %)/100) = Total estimated gas with  
premium (Gwei)  
Total estimated gas with premium (Gwei) / 1,000,000,000 Gwei/ETH = Total  
estimated gas with premium (ETH)  
Total estimated gas with premium (ETH) / (ETH/LINK price) = Total estimated gas  
with premium (LINK)
```

```
</Fragment>  
<Fragment slot="panel.2">
```

```
((Gas lane maximum (Max verification gas + Callback gas limit)) (100 +  
premium %)/100) / (1,000,000,000 Gwei/ETH)) / (ETH/[Native token] price) =  
Minimum [Native token]
```

Here is the same formula, broken out into steps:

```
Gas lane maximum (Max verification gas + Callback gas limit) = Total estimated  
gas (Gwei)  
Total estimated gas (Gwei) ((100 + premium %)/100) = Total estimated gas with  
premium (Gwei)  
Total estimated gas with premium (Gwei) / 1,000,000,000 Gwei/ETH = Total  
estimated gas with premium (ETH)  
Total estimated gas with premium (ETH) / (ETH/[Native token] price) = Total  
estimated gas with premium (Native token)
```


</Fragment>
</TabsContent>

Maximum consuming contracts

Each subscription supports up to 100 consuming contracts. If you need more than 100 consuming contracts, create multiple subscriptions.

Coordinator contract limits

You can see the configuration for each network on the Supported networks page. You can also view the full configuration for each coordinator contract directly in the block explorer for that network, for example, Etherscan or Polygonscan.

- Each coordinator has a MAXNUMWORDS parameter that limits the maximum number of random values you can receive in each request.
- Each coordinator has a maxGasLimit parameter, which is the maximum allowed callbackGasLimit value for your requests. You must specify a sufficient callbackGasLimit to fund the callback request to your consuming contract. This depends on the number of random values you request and how you process them in your fulfillRandomWords() function. If your callbackGasLimit is not sufficient, the callback fails but your subscription is still charged for the work done to generate your requested random values.

create-manage.mdx:

```
---  
section: vrf  
date: Last Modified  
title: "Create and manage VRF V2.5 subscriptions"  
---  
  
import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"  
import { CodeSample } from "@components"  
import { Tabs, TabsContent } from "@components/Tabs"  
  
<Vrf25Common callout="security" />
```

Using the VRF Subscription Manager

The VRF Subscription Manager is available to help you create and manage VRF V2.5 subscriptions. You can create and manage new V2.5 subscriptions, and manage existing V2 subscriptions, but you can no longer create new V2 subscriptions in the VRF Subscription Manager. Alternatively, you can create and manage subscriptions programmatically.

Create a subscription

To create a VRF 2.5 subscription:

1. Use the VRF Subscription Manager at vrf.chain.link. Connect your wallet in the upper right corner and then click Create subscription. The address of your connected wallet is automatically filled in the Admin address field.

1. When the subscription is successfully created, there will be an alert in the upper right corner telling you that the subscription was successfully created. Click Home to navigate back to your main dashboard.

1. Your new subscription shows in the My Subscriptions list, along with previous V2 subscriptions you might have. Click the Subscription ID for your new subscription in the list.

Add a consumer

To add a consuming contract:

1. On the details page for your subscription, select Add Consumer.

1. Provide the address of your consuming contract, and then select Add Consumer again. Confirm the resulting prompts in MetaMask or other wallet browser extension.

Fund your subscription

To fund your subscription:

1. On the page for your subscription, select the Actions menu and then select Fund subscription.

1. Your subscription has two balances: one for LINK, and one for the native token. Expand the Asset menu to select either LINK or the native token.

1. In Amount to fund, enter the amount you want to fund your subscription. Your wallet balance is displayed below the Asset field for easier reference. Select Confirm to fund your wallet, and then confirm the resulting prompts in MetaMask or other wallet browser extension.

Create a subscription programmatically

If you prefer to create, fund and manage your subscriptions programmatically, you can either deploy a subscription manager contract or use your network's block explorer:

1. Create a new subscription for VRF v2.5:

```
{/ prettier-ignore /}  
<TabsContent sharedStore="subCreationMethod" client:visible>  
  <Fragment slot="tab.1">Subscription contract</Fragment>  
  <Fragment slot="tab.2">Using a block explorer</Fragment>  
  <Fragment slot="panel.1">  
    Deploy the SubscriptionManager contract. On deployment, the contract creates  
a new subscription and adds itself as a consumer to the new subscription.  
  </Fragment>  
  <Fragment slot="panel.2">  
    1. Navigate to the VRF coordinator contract on the block explorer for the  
network you want to use (for example, Etherscan or Polygonscan). You can find  
the coordinator addresses with links to the block explorers on the Supported  
Networks page.  
    1. In the Contract tab, select the Write contract tab. Connect your wallet  
to the block explorer.  
    1. Expand the createSubscription function and select the Write button.  
Follow the prompts in MetaMask to confirm the transaction.  
    1. Get your subscription ID for the next step, funding your subscription.  
  </Fragment>  
</TabsContent>
```

1. Fund your new VRF v2.5 subscription:

```
{/ prettier-ignore /}  
<TabsContent sharedStore="subCreationMethod" client:visible>  
  <Fragment slot="tab.1">Subscription contract</Fragment>  
  <Fragment slot="tab.2">Using a block explorer</Fragment>  
  <Fragment slot="panel.1">  
    1. Fund your new VRFv2PlusSubscriptionManager contract.  
    1. Call topUpSubscription from the VRFv2PlusSubscriptionManager contract.
```

This function uses the LINK token contract's transferAndCall function to fund your new subscription.

</Fragment>

<Fragment slot="panel.2">

1. Fund your new VRFv2PlusSubscriptionManager contract.

1. Navigate to the LINK token contract on the block explorer for the network you want to use (for example, Etherscan or Polygonscan). You can find the LINK token contract addresses with links to the block explorers on the Supported Networks page.

1. In the Contract tab, select the Write contract tab. Connect your wallet to the block explorer.

1. Expand the transferAndCall function and fill in the following parameters:

- to(address): The address of the VRF coordinator.
- value(uint256): The amount you want to fund your subscription with.
- data(bytes): The ABI-encoded subscription ID.

1. Select the Write button and follow the prompts in MetaMask to confirm the transaction.

</Fragment>

</TabsContent>

get-a-random-number.mdx:

section: vrf

date: Last Modified

title: "Get a Random Number"

whatsnext:

```
{
  "Security Considerations": "/vrf/v2-5/security",
  "Best Practices": "/vrf/v2-5/best-practices",
  "Migrating from V2": "/vrf/v2-5/migration-from-v2",
  "Supported Networks": "/vrf/v2-5/supported-networks",
}
```

metadata:

description: "How to generate a random number inside a smart contract using Chainlink VRF V2.5."

import Vrf25Common from "@features/vrf/v2-5/Vrf25Common.astro"

import { Aside, CodeSample } from "@components"

This guide explains how to get random values using a simple contract to request and receive random values from Chainlink VRF v2.5. The guide uses the Subscription Manager to create and manage your subscription. Alternatively, you can also create and manage subscriptions programmatically. To explore more applications of VRF, refer to our blog.

<Vrf25Common callout="uicallout" />

Requirements

This guide assumes that you know how to create and deploy smart contracts on Ethereum testnets using the following tools:

- The Remix IDE
- MetaMask
- Sepolia testnet ETH

If you are new to developing smart contracts on Ethereum, see the Getting Started guide to learn the basics.

Create and fund a subscription

For this example, create a new subscription on the Sepolia testnet.

1. Open MetaMask and set it to use the Sepolia testnet. The Subscription Manager detects your network based on the active network in MetaMask.

1. Check MetaMask to make sure you have testnet ETH and LINK on Sepolia. You can get testnet ETH and LINK at faucets.chain.link/sepolia.

1. Open the Subscription Manager at vrf.chain.link.

```
{/ prettier-ignore /}
```

```
<div class="remix-callout">
  <a href="https://vrf.chain.link">Open the Subscription Manager</a>
</div>
```

1. Click Create Subscription and follow the instructions to create a new subscription account. If you connect your wallet to the Subscription Manager, the Admin address for your subscription is prefilled and not editable. Optionally, you can enter an email address and a project name for your subscription, and both of these are private. MetaMask opens and asks you to confirm payment to create the account onchain. After you approve the transaction, the network confirms the creation of your subscription account onchain.

1. After the subscription is created, click Add funds and follow the instructions to fund your subscription.

For your request to go through, you need to fund your subscription with enough testnet funds to meet your minimum subscription balance to serve as a buffer against gas volatility.

- If you're paying with testnet LINK, fund your contract with 7 LINK. (After your request is processed, the actual cost will be around 0.06 LINK, and that amount will be deducted from your subscription balance.)

- If you're paying with testnet ETH, fund your contract with 0.03 ETH. (After your request is processed, the actual cost will be around 0.000247 ETH, and that amount will be deducted from your subscription balance.)

MetaMask opens to confirm the token transfer to your subscription. After you approve the transaction, the network confirms the transfer of your testnet funds to your subscription account.

1. After you add funds, click Add consumer. A page opens with your account details and subscription ID.

1. Record your subscription ID, which you need for your consuming contract. You will add the consuming contract to your subscription later.

You can always find your subscription IDs, balances, and consumers at vrf.chain.link.

Now that you have a funded subscription account and your subscription ID, create and deploy a VRF compatible contract.

Create and deploy a VRF compatible contract

For this example, use the SubscriptionConsumer.sol sample contract. This contract imports the following dependencies:

- VRFConsumerBaseV2Plus.sol(link)
- VRFV2PlusClient.sol(link)

The contract also includes pre-configured values for the necessary request parameters such as vrfCoordinator address, gas lane keyHash, callbackGasLimit,

requestConfirmations and number of random words numWords. You can change these parameters if you want to experiment on different testnets, but for this example you only need to specify subscriptionId when you deploy the contract.

Build and deploy the contract on Sepolia.

1. Open the SubscriptionConsumer.sol in Remix.

```
{/ prettier-ignore /}  
<CodeSample src="samples/VRF/v2-5/SubscriptionConsumer.sol" showButtonOnly/>
```

1. On the Compile tab in Remix, compile the SubscriptionConsumer.sol contract.

1. Configure your deployment. On the Deploy tab in Remix, select the Injected Provider environment, select the SubscriptionConsumer contract from the contract list, and specify your subscriptionId so the constructor can set it.

!Example showing the deploy button with the subscriptionID field filled in Remix

1. Click the Deploy button to deploy your contract onchain. MetaMask opens and asks you to confirm the transaction.

1. After you deploy your contract, copy the address from the Deployed Contracts list in Remix. Before you can request randomness from VRF v2.5, you must add this address as an approved consuming contract on your subscription account.

!Example showing the contract address listed under the Contracts list in Remix

1. Open the Subscription Manager at vrf.chain.link and click the ID of your new subscription under the My Subscriptions list. The subscription details page opens.

1. Under the Consumers section, click Add consumer.

1. Enter the address of your consuming contract that you just deployed and click Add consumer. MetaMask opens and asks you to confirm the transaction.

Your example contract is deployed and approved to use your subscription balance to pay for VRF v2.5 requests. Next, request random values from Chainlink VRF.

Request random values

The deployed contract requests random values from Chainlink VRF, receives those values, builds a struct RequestStatus containing them and stores the struct in a mapping srequests. Run the requestRandomWords() function on your contract to start the request.

1. Return to Remix and view your deployed contract functions in the Deployed Contracts list.

1. Expand the requestRandomWords() function to send the request for random values to Chainlink VRF. Use enableNativePayment to specify whether you want to pay in native tokens or LINK:

- To use native tokens, set enableNativePayment to true.
- To use LINK, set enableNativePayment to false.

When you click transact, MetaMask opens and asks you to confirm the transaction. After you approve the transaction, Chainlink VRF processes your request. Chainlink VRF fulfills the request and returns the random values to your contract in a callback to the fulfillRandomWords() function. At this point, a new key requestId is added to the mapping srequests.

Depending on current testnet conditions, it might take a few minutes for the callback to return the requested random values to your contract. You can see a list of pending requests for your subscription ID at `vrf.chainlink`.

1. To fetch the request ID of your request, call `lastRequestId()`.

1. After the oracle returns the random values to your contract, the mapping `srequests` is updated: The received random values are stored in `srequests[requestId].randomWords`.

1. Call `getRequestStatus()` specifying the `requestId` to display the random words.

You deployed a simple contract that can request and receive random values from Chainlink VRF. Next, learn how to create and manage subscriptions programmatically by using a smart contract instead of the Subscription Manager.

<Aside type="note" title="Note on Requesting Randomness">

Do not allow re-requesting or cancellation of randomness. For more information, see the VRF Security Considerations page.

</Aside>

Analyzing the contract

In this example, your MetaMask wallet is the subscription owner and you created a consuming contract to use that subscription. The consuming contract uses static configuration parameters.

<CodeSample src="samples/VRF/v2-5/SubscriptionConsumer.sol" />

The parameters define how your requests will be processed. You can find the values for your network in the Configuration page.

- `uint256 sssubscriptionId`: The subscription ID that this contract uses for funding requests.

- `bytes32 keyHash`: The gas lane key hash value, which is the maximum gas price you are willing to pay for a request in wei. It functions as an ID of the offchain VRF job that runs in response to requests.

- `uint32 callbackGasLimit`: The limit for how much gas to use for the callback request to your contract's `fulfillRandomWords()` function. It must be less than the `maxGasLimit` limit on the coordinator contract. Adjust this value for larger requests depending on how your `fulfillRandomWords()` function processes and stores the received random values. If your `callbackGasLimit` is not sufficient, the callback will fail and your subscription is still charged for the work done to generate your requested random values.

- `uint16 requestConfirmations`: How many confirmations the Chainlink node should wait before responding. The longer the node waits, the more secure the random value is. It must be greater than the `minimumRequestBlockConfirmations` limit on the coordinator contract.

- `uint32 numWords`: How many random values to request. If you can use several random values in a single callback, you can reduce the amount of gas that you spend per random value. The total cost of the callback request depends on how your `fulfillRandomWords()` function processes and stores the received random values, so adjust your `callbackGasLimit` accordingly.

The contract includes the following functions:

- `requestRandomWords(bool enableNativePayment)`: Takes your specified parameters and submits the request to the VRF coordinator contract. Use `enableNativePayment`

to specify for each request whether you want to pay in native tokens or LINK:

- To use native tokens, set `enableNativePayment` to `true`.
- To use LINK, set `enableNativePayment` to `false`.
- `fulfillRandomWords()`: Receives random values and stores them with your contract.
- `getRequestStatus()`: Retrieve request details for a given `requestId`.

```
<Aside type="note" title="Security Considerations">  
  Be sure to review your contracts to make sure they follow the best practices  
  on the security  
  considerations page.  
</Aside>
```

Clean up

After you are done with this contract and the subscription, you can retrieve the remaining testnet tokens to use with other examples.

1. Open the Subscription Manager at vrf.chain.link and click the ID of your new subscription under the My Subscriptions list. The subscription details page opens.

1. On your subscription details page, expand the Actions menu and select Cancel subscription. A field displays, prompting you to add the wallet address you want to send the remaining funds to.

1. Enter your wallet address and click Cancel subscription. MetaMask opens and asks you to confirm the transaction. After you approve the transaction, Chainlink VRF closes your subscription account and sends the remaining LINK to your wallet.

test-locally.mdx:

```
---  
section: vrf  
title: "Local testing using a mock subscription contract"  
metadata:  
  description: "Example contract for generating random words using the VRF v2.5  
subscription method on your local blockchain using a mock contract."  
---
```

```
import VrfCommon from "@features/vrf/v2/common/VrfCommon.astro"  
import ContentCommon from "@features/common/ContentCommon.astro"  
import { Aside, ClickToZoom, CodeSample, CopyText, Icon } from "@components"  
import { linkEth } from "@features/data"  
import { LatestPrice } from "@features/feeds"  
import button from "@chainlink/design-system/button.module.css"
```

This guide explains how to test Chainlink VRF v2.5 on a Remix IDE sandbox blockchain environment. Note: You can reuse the same logic on another development environment, such as Hardhat or Foundry. For example, read the Hardhat Starter Kit `RandomNumberConsumer` unit tests.

```
<Aside type="caution" title="Test on public testnets thoroughly">  
  Even though local testing has several benefits, testing with a VRF mock covers  
  the bare minimum of use cases. Make  
  sure to test your consumer contract thoroughly on public testnets.  
</Aside>
```

Benefits of local testing

<ContentCommon section="localTestingBenefits" />

Testing logic

Complete the following tasks to test your VRF v2.5 consumer locally:

1. Deploy the VRFCoordinatorV25Mock. This contract is a mock of the VRFCoordinatorV25 contract.
1. Call the createSubscription function (which VRFCoordinatorV25Mock inherits) to create a new subscription.
1. Call the VRFCoordinatorV25Mock fundSubscription function to fund your newly created subscription. Note: You can fund with an arbitrary amount.
1. Deploy your VRF consumer contract.
1. Call the addConsumer function (which VRFCoordinatorV25Mock inherits) to add your consumer contract to your subscription.
1. Request random words from your consumer contract.
1. Call the VRFCoordinatorV25Mock fulfillRandomWords function to fulfill your consumer contract request.

Testing

Open the contracts on Remix IDE

For local testing, use the default "Remix VM" environment.

Open VRFv25Consumer and compile in Remix:

<CodeSample src="samples/VRF/mock/VRFv25Consumer.sol" />

Open VRFCoordinatorV25Mock in Remix:

<CodeSample src="samples/VRF/mock/VRFCoordinatorV25Mock.sol" />

On the Solidity Compiler tab, expand the Advanced Configurations section and check the Enable optimization box before you compile the VRFCoordinatorV25Mock contract:

<ClickToZoom src="/images/vrf/v2-5/mock/enable-optimization.png" />

Your Remix IDE file explorer should display VRFCoordinatorV25Mock.sol and VRFv25Consumer.sol:

<ClickToZoom src="/images/vrf/v2-5/mock/file-explorer.png" />

Deploy VRFCoordinatorV25Mock

1. Open VRFCoordinatorV25Mock.sol.

1. Under DEPLOY & RUN TRANSACTIONS, select VRFCoordinatorV25Mock.

<ClickToZoom src="/images/vrf/v2-5/mock/deployment-contracts-list.png" />

1. Under DEPLOY, fill in the BASEFEE, GASPRICELINK and WEIPERUNITLINK. These variables are used in the VRFCoordinatorV25Mock contract to represent the base fee, the gas price (in LINK tokens), and the current LINK/ETH price for the VRF requests.

<ClickToZoom src="/images/vrf/v2-5/mock/mock-deployment-parameters.png" />

You can set:

- BASEFEE to <CopyText text="1000000000000000000" code/>
- GASPRICELINK to <CopyText text="1000000000" code/>

- WEIPERUNITLINK to the current LINK/ETH price. Click the "Latest Price" button to view it:

```
{ " " }  
<LatestPrice client:idle feedAddress={linkEth.link.eth.sepolia.address}  
supportedChain="ETHEREUMSEPOLIA" />
```

1. Click transact to deploy the VRFCoordinatorV25Mock contract.

1. Once deployed, you should see the VRFCoordinatorV25Mock contract under Deployed Contracts.

```
<ClickToZoom src="/images/vrf/v2-5/mock/deployed-mock.png" />
```

1. Note the address of the deployed contract.

Create and fund a subscription

1. Click createSubscription to create a new subscription.

1. In the Remix IDE console, read your transaction decoded output to find the subscription ID. Note the subscription ID, which is required for multiple steps in this tutorial.

```
<ClickToZoom src="/images/vrf/v2-5/mock/example-output-sub-id.png" />
```

1. Click on fundSubscription to fund your subscription. Fill in your subscription ID for subid and set the amount to `<CopyText text="100000000000000000000" code/>`. This mocks funding your subscription with 100 LINK.

Deploy the VRF consumer contract

1. In the file explorer, open VRFv25Consumer.sol.

1. Under DEPLOY & RUN TRANSACTIONS, select RandomNumberConsumerV25.

```
<ClickToZoom src="/images/vrf/v2-5/mock/deploy-consumer.png" />
```

1. Under DEPLOY, fill in the following parameters:

- SUBSCRIPTIONID with your subscription ID
- VRFCOORDINATOR with the deployed VRFCoordinatorV25Mock address
- KEYHASH with an arbitrary bytes32 (In this example, you can set the KEYHASH to `<CopyText text="0x787d74caea10b2b357790d5b5247c2f63d1d91572a9846f780606e4d953677ae" code/>`).

1. Click transact to deploy the RandomNumberConsumerV25 contract.

1. After the consumer contract is deployed, you should see the RandomNumberConsumerV25 contract under Deployed Contracts. Note the address of the deployed contract.

Add the consumer contract to your subscription

1. Under Deployed Contracts, open the functions list of your deployed VRFCoordinatorV25Mock contract.

1. Click addConsumer and fill in the subid with your subscription ID and consumer with your deployed consumer contract address.

```
<ClickToZoom src="/images/vrf/v2-5/mock/add-consumer.png" />
```

1. Click transact.

Request random words

1. Under Deployed Contracts, open the functions list of your deployed RandomNumberConsumerV25 contract.

1. Click requestRandomWords.

```
<ClickToZoom src="/images/vrf/mock/v2-requestrandomwords.jpg" />
```

1. Click srequestId to display the last request ID. In this example, the output is 1.

```
<ClickToZoom src="/images/vrf/v2-5/mock/show-last-request-id.png" />
```

1. Note your request ID.

Fulfill the VRF request

Because you are testing on a local blockchain environment, you must fulfill the VRF request yourself.

1. Under Deployed Contracts, open the functions list of your deployed VRFCoordinatorV25Mock contract.

1. Click fulfillRandomWords and fill in requestId with your VRF request ID and consumer with your consumer contract address.

```
<ClickToZoom src="/images/vrf/v2-5/mock/manual-fulfill-request.png" />
```

1. Click transact.

Check the results

1. Under Deployed Contracts, open the functions list of your deployed RandomNumberConsumerV25 contract.

1. For each VRF request, your consumer contract requests two random words. After a request is fulfilled, the two random words are stored in the srandomWords array. You can check the stored random words by reading the two first indexes of the srandomWords array. To do so, click the srandomWords function and:

1. Fill in the index with 0 then click call to read the first random word.

```
{" "}
```

```
<ClickToZoom src="/images/vrf/v2-5/mock/show-random-word.png" />
```

1. You can read the second random word in a similar way: fill in the index with 1 then click call to display the second random word.

Next steps

This guide demonstrated how to test a VRF v2.5 consumer contract on your local blockchain. The guide uses the Remix IDE for learning purposes, but you can reuse the same testing logic in another development environment, such as Hardhat. For example, see the Hardhat Starter Kit RandomNumberConsumer unit tests.