Getting Started with Chainlink Data Streams using the Hardhat CLI

Mainnet Access

Chainlink Data Streams is available on Arbitrum Mainnet and Arbitrum Sepolia.

Talk to an expert

Contact us to talk to an expert about integrating Chainlink Data Streams with your applications.

This guide shows you how to read data from a Data Streams feed, verify the answer onchain, and store it. This CLI guide uses the Hardhat Framework so you can complete these steps using terminal commands rather than the web-based Remix IDE. If you prefer Remix or are unfamiliar with how to run terminal commands, read the Getting Started - Remix IDE guide instead.

This example uses a Chainlink Automation Log Trigger to check for events that require data. The flow follows this sequence:

- A simple emitter contract emits a log that triggers the upkeep.
- Chainlink Automation then uses StreamsLookup to retrieve a signed report from the Data Streams Engine, returns the data in a callback, and runs the performUpkeep function on your registered upkeep contract.
- The performUpkeep function calls the verify function on the verifier contract and stores the retrieved price onchain.

Disclaimer

This guide 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.

Before you begin

This guide uses the Hardhat development environment to deploy and interact with the contracts. To learn more about Hardhat, read the Hardhat Documentation .

Requirements

- Git: Make sure you have Git installed. You can check your current version by running git --version in your terminal and download the latest version from the official Git website if necessary.
- Node.js and npm: Install the latest release of Node.js 20. Optionally, you can use the nvm package to switch between Node.js versions with nvm use 20. To ensure you are running the correct version in a terminal, type node -v. $node -v v20.11.0
- Testnet funds: This guide requires testnet ETH and LINK on Arbitrum Sepolia. Both are available at faucets.chain.link .

Tutorial

Setup

1. Clone the repository that contains the Hardhat project setup for this guide. This repository contains the Solidity contracts and the Hardhat configuration files you need to deploy and interact with the contracts.

git clone https://github.com/smartcontractkit/smart-contract-examples.git cd data-streams/getting-started/hardhat 2. Install all the dependencies:

npm install 3. Set an encryption password for your environment variables. This password needs to be set each time you create or restart a terminal shell session.

npx env-enc set-pw 4. Set the required environment variables using the following command:

npx env-enc set * PRIVATE_KEY: The private key for your testnet wallet that will deploy and interact with the contracts. If you use MetaMask, follow the instructions to Export a Private Key . * ARBITRUM_SEPOLIA_RPC_URL: 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.

Deploy the upkeep and the log emitter contracts

Deploy an upkeep contract that is enabled to retrieve data from Data Streams. For this example, you will read from the ETH/USD Data Streams feed with ID 0x00027bbaff688c906a3e20a34fe951715d1018d262a5b66e38eda027a674cd1b on Arbitrum Sepolia. See the Data Streams Feed IDs page for a complete list of available assets, IDs, and verifier proxy addresses.

Execute the following command to deploy the Chainlink Automation upkeep contract and the Log Emitter contract to the Arbitrum Sepolia network.

npx hardhat deployAll --network arbitrumSepolia Expect output similar to the following in your terminal:

ℹ Deploying StreamsUpkeepRegistrar contract... ✔ StreamsUpkeepRegistrar deployed at: 0x48403478Aa021A9BC30Da0BDE47cbc155CcA8916 ℹ Deploying LogEmitter contract... ✔ LogEmitter deployed at: 0xD721337a827F9D814daEcCc3c7e72300af914BFE ✔ All contracts deployed successfully. Save the deployed contract addresses for both contracts. You will use these addresses later.

Fund the upkeep contract

In this example, the upkeep contract pays for onchain verification of reports from Data Streams. The Automation subscription does not cover the cost. Transfer 1.5 testnet LINK to the upkeep contract address you saved earlier. You can retrieve unused LINK later.

npx hardhat transfer-link --recipient --amount 1500000000000000000 --network arbitrumSepolia Replace with the address of the StreamsUpkeepRegistrar contract you saved earlier.

Expect output similar to the following in your terminal:

ℹ Starting LINK transfer from to the streams upkeep contract at 0xD721337a827F9D814daEcCc3c7e72300af914BFE ℹ LINK token address: 0xb1D4538B4571d411F07960EF2838Ce337FE1E80E ℹ LINK balance of sender 0x45C90FBb5acC1a5c156a401B56Fea55e69E7669d is 6.5 LINK ✔ 1.5 LINK were sent from 0x45C90FBb5acC1a5c156a401B56Fea55e69E7669d to 0xD721337a827F9D814daEcCc3c7e72300af914BFE. Transaction Hash: 0xf241bf4415ec081325ccd8ec3d54432e424afd16f1c81fa78b291ae9a0c03ce2

Register and fund the upkeep

Programmatically register and fund a new Log Trigger upkeep with 1 LINK:

npx hardhat registerAndFundUpkeep --streams-upkeep --log-emitter --network arbitrumSepolia Replace and with the addresses of your StreamsUpkeepRegistrar and LogEmitter contracts.

Expect output similar to the following in your terminal:

✔ Upkeep registered and funded with 1 LINK successfully.

Emit a log

Now, you can use your emitter contract to emit a log and initiate the upkeep, which retrieves data for the specified Data Streams feed ID.

npx hardhat emitLog --log-emitter --network arbitrumSepolia Replace with the address of your LogEmitter contract.

Expect output similar to the following in your terminal:

✔ Log emitted successfully in transaction: 0x236ee95faade12d1b6d497ee2e51ddf957f7d4986ffe51d784b923081ed440ff After the transaction is complete, the log is emitted, and the upkeep is triggered.

View the retrieved price

The retrieved price is stored in the s_last_retrieved_price contract variable and emitted in the logs. To see the price retrieved by the StreamsUpkeepRegistrar contract:

npx hardhat getLastRetrievedPrice --streams-upkeep --network arbitrumSepolia Replace with the address of your StreamsUpkeepRegistrar contract.

Expect output similar to the following in your terminal:

✔ Last Retrieved Price: 2945878120219995000000 The answer on the ETH/USD feed uses 18 decimal places, so an answer of 2945878120219995000000 indicates an ETH/USD price of 2945.878120219995. Each Data Streams feed uses a different number of decimal places for answers. See the Data Streams Feed IDs page for more information.

Alternatively, you can view the price emitted in the logs for your upkeep transaction.

You can find the upkeep transaction hash atChainlink Automation UI and view the transaction logs in theArbitrum Sepolia explorer.

Examine the code

The example code you deployed has all the interfaces and functions required to work with Chainlink Automation as an upkeep contract. It follows a similar flow to the trading flow in thisArchitecture documentation but uses a basic log emitter to simulate the client contract that would initiate aStreamsLookup. The code example usesrevertwithStreamsLookupto convey call information about what streams to retrieve. See theEIP-3668 rationale for more information about how to userevertin this way.

// SPDX-License-Identifier: MITpragmasolidity0.8.19;import{Common}from"@chainlink/contracts/src/v0.8/llo-feeds/libraries/Common.sol";import{StreamsLookupCompatibleInterface}from"@chainlink/contracts/src/v0.8/automation/interfaces/StreamsLookupCompatibleInterface.sol";import{ILogAutomation,Log}fro feeds/interfaces/IRewardManager.sol";import{IVerifierFeeManager}from"@chainlink/contracts/src/v0.8/llo-feeds/interfaces/IVerifierFeeManager.sol";import{IERC20}from"@chainlink/contracts/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/interfaces/IERC20.sol";import{LinkTokenInterface}from"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.sol";/ *** THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE FOR DEMONSTRATION PURPOSES. * DO NOT USE THIS CODE IN PRODUCTION. ***// * @dev Defines the parameters required to register a new upkeep. * @param name The name of the upkeep to be registered. * @param encryptedEmail An encrypted email address associated with the upkeep (optional). * @param upkeepContract The address of the contract that requires upkeep. * @param gasLimit The maximum amount of gas to be used for the upkeep execution. * @param adminAddress The address that will have administrative privileges over the upkeep. * @param triggerType An identifier for the type of trigger that initiates the upkeep (1 for event-based). * @param checkData Data passed to the checkUpkeep function to simulate conditions for triggering upkeep. * @param triggerConfig Configuration parameters specific to the trigger type. * @param offchainConfig Off-chain configuration data, if applicable. * @param amount The amount of LINK tokens to fund the upkeep registration.

/structRegistrationParams{stringname;bytesencryptedEmail;addressupkeepContract;uint32gasLimit;addressadminAddress;uint8triggerType;bytescheckData;bytestriggerConfig;bytesoffchainConfig;uint9

** @dev Interface for the Automation Registrar contract.* /interfaceAutomationRegistrarInterface{/ *** @dev Registers a new upkeep contract with Chainlink Automation. * @param requestParams The parameters required for the upkeep registration, encapsulated in RegistrationParams. * @return upkeepID The unique identifier for the registered upkeep, used for future interactions. */functionregisterUpkeep(RegistrationParamscalldatarequestParams)externalreturns(uint256);}// Custom interfaces for Data Streams: IVerifierProxy and IFeeManagerinterfaceIVerifierProxy{functionverify(bytescalldatapayload,bytescalldataparameterPayload)externalpayablereturns(bytesmemoryverifierResponse);functions_feeManager()ext i_link;AutomationRegistrarInterfacepublicimmutable i_registrar;structBasicReport{bytes32feedId;// The feed ID the report has data foruint32validFromTimestamp;// Earliest timestamp for which price is applicableuint32observationsTimestamp;// Latest timestamp for which price is applicableuint192nativeFee;// Base cost to validate a transaction using the report, denominated in the chain's native token (WETH/ETH)uint192linkFee;// Base cost to validate a transaction using the report, denominated in LINKuint32expiresAt;// Latest timestamp where the report can be verified onchainint192price;// DON consensus median price, carried to 8 decimal places}structPremiumReport{bytes32feedId;// The feed ID the report has data foruint32validFromTimestamp;// Earliest timestamp for which price is applicableuint32observationsTimestamp;// Latest timestamp for which price is applicableuint192nativeFee;// Base cost to validate a transaction using the report, denominated in the chain's native token (WETH/ETH)uint192linkFee;// Base cost to validate a transaction using the report, denominated in LINKuint32expiresAt;// Latest timestamp where the report can be verified onchainint192price;// DON consensus median price, carried to 8 decimal placesint192bid;// Simulated price impact of a buy order up to the X% depth of liquidity utilisationint192ask;// Simulated price impact of a sell order up to the X% depth of liquidity utilisation}structQuote{addressquoteAddress;}eventPriceUpdate(int192indexedprice);IVerifierProxypublicverifier;addresspublicFEE_ADDRESS;stringpublicconstantDATASTREAMS_FEED Find a complete list of IDs at https://docs.chain.link/data-streams/stream-idsstring[]publicfeedIds;constructor(address_verifier,LinkTokenInterface link,AutomationRegistrarInterface registrar,string[]memory_feedIds){verifier=IVerifierProxy(_verifier);i_link=link;i_registrar=registrar;feedIds=_feedIds;}/** * @notice Registers a new upkeep using the specified parameters and predicts its ID. * @dev This function first approves the transfer of LINK tokens specified in params.amount to the Automation Registrar contract. * It then registers the upkeep and stores its ID if registration is successful. * Reverts if auto-approve is disabled or registration fails. * @param params The registration parameters, including name, upkeep contract address, gas limit, admin address, trigger type, and funding amount.

*/functionregisterAndPredictID(RegistrationParamsmemoryparams)public{i_link.approve(address(i_registrar),params.amount);uint256upkeepID=i_registrar.registerUpkeep(params);if(upkeepID!=0) {s_upkeepID=upkeepID;// DEV - Use the upkeepID however you see fit}else{revert("auto-approve disabled");}}/ * @notice this is a new, optional function in streams lookup. It is meant to surface streams lookup errors. * @return upkeepNeeded boolean to indicate whether the keeper should call performUpkeep or not. * @return performData bytes that the keeper should call performUpkeep with, if * upkeep is needed. If you would like to encode data to decode later, try* abi.encode*.*

/functioncheckErrorHandler(uint256/*errCode*/,bytesmemory/*extraData*/)externalpurereturns(boolupkeepNeeded,bytesmemoryperformData){return(true,"0");// Hardcoded to always perform upkeep.// Read the StreamsLookup error handler guide for more information.// https://docs.chain.link/chainlink-automation/guides/streams-lookup-error-handler}// This function uses revert to convey call information.// See https://eips.ethereum.org/EIPS/eip-3668#rationale for details.functioncheckLog(Logcalldatalog,bytesmemory)externalreturns(boolupkeepNeeded,bytesmemoryperformData) {revertStreamsLookup(DATASTREAMS_FEEDLABEL,feedIds,DATASTREAMS_QUERYLABEL,log.timestamp,"");}// The Data Streams report bytes is passed here.// extraData is context data from feed lookup process.// Your contract may include logic to further process this data.// This method is intended only to be simulated offchain by Automation.// The data returned will then be passed by Automation into performUpkeepfunctioncheckCallback(bytes[]calldatavalues,bytescalldataextraData)externalpurereturns(bool,bytesmemory){return(true,abi.encode(values,extraData));}// function will be performed onchainfunctionperformUpkeep(bytescalldataperformData)external{// Decode the performData bytes passed in by CL Automation.// This contains the data returned by your implementation in checkCallback().(bytes[]memorysignedReports,bytesmemoryextraData)=abi.decode(performData,(bytes[],bytes));bytesmemoryunverifiedReport=signedReports[0];(,/* *bytes32[3] reportContextData* /bytesmemoryreportData)=abi.decode(unverifiedReport,(bytes32[3],bytes));// Report verification feesIFeeManager feeManager=IFeeManager(address(verifier.s_feeManager()));IRewardManager rewardManager=IRewardManager(address(feeManager.i_rewardManager()));addressfeeTokenAddress=feeManager.i_linkAddress(); (Common.Assetmemoryfee,,)=feeManager.getFeeAndReward(address(this),reportData,feeTokenAddress);// Approve rewardManager to spend this contract's balance in feesIERC20(feeTokenAddress).approve(address(rewardManager),fee.amount);// Verify the reportbytesmemoryverifiedReportData=verifier.verify(unverifiedReport,abi.encode(feeTokenAddress));// Decode verified report data into BasicReport structBasicReportmemoryverifiedReport=abi.decode(verifiedReportData,(BasicReport));// Log price from reportemitPriceUpdate(verifiedReport.price);// Store the price from the reports_last_retrieved_price=verifiedReport.price;}} Open in Remix What is Remix?

Initializing the contract

When deploying the contract, you define:

1. The verifier proxy address for the Data Streams feed you want to read from. You can find this address on theData Streams Feed IDs page. The verifier proxy address provides functions that are required for this example:
2. Thes_feeManagerfunction to estimate the verification fees.
3. Theverifyfunction to verify the report onchain.
4. The LINK token address. This address is used to register and fund your upkeep. You can find the LINK token address on theChainlink Token Addresses page.
5. The registrar's contract address. This address is used to register your upkeep. You can find the registrar contract addresses on theChainlink Automation Supported Networks page.

Funding the upkeep contract

In this example, you must fund theStreamsUpkeepRegistrarcontract with testnet LINK tokens to pay the onchain report verification fees. You can use thetransfer-link task to transfer LINK tokens to theStreamsUpkeepRegistrarcontract you deployed.

Thetransfer-linkHardhat task sets up the necessary parameters for the LINK token transfer and submits the transfer request to the LINK token contract using thetransferfunction.

Note:Funding theStreamsUpkeepRegistrarcontract is distinct from funding your Chainlink Automation upkeep to pay the fees to perform the upkeep.

Registering the upkeep

You need to register your log-triggered upkeep with the Chainlink Automation registrar. You can use theegisterAndFundLogUpkeep task to programmatically register theStreamsUpkeepRegistrarandLogEmittercontracts with the Chainlink Automation registrar. The task also funds the upkeep with 1 testnet LINK token.

TheregisterAndFundLogUpkeepHardhat task sets up the necessary parameters for upkeep registration, including trigger configuration for a Log Emitter contract, and submits the registration request to the registrar contract via theregisterAndPredictIDfunction.

You can use theChainlink Automation UI to view the registered upkeep and the upkeep's configuration.

Emitting a log, retrieving, and verifying the report

You can use theemitLogtask to emit a log from theLogEmittercontract.

1. The emitted log triggers the Chainlink Automation upkeep.
2. Chainlink Automation then usesStreamsLookupto retrieve a signed report from the Data Streams Engine, returns the data in a callback (checkCallback), and runs theperformUpkeepfunction on your registered upkeep contract.
3. TheperformUpkeepfunction calls theverifyfunction on the verifier contract to verify the report onchain.
4. In this example, theperformUpkeepfunction also stores the price from the report in thes_last_retrieved_pricestate variable and emits aPriceUpdatelog message with the price.

Viewing the retrieved price

ThegetLastRetrievedPrice Hardhat task retrieves the last price updated by theperformUpkeepfunction in thes_last_retrieved_pricestate variable of theStreamsUpkeepRegistrarcontract.

Optional: Handle Data Streams fetching errors offchain withcheckErrorHandler

When Automation detects the triggering event, it runs thecheckLogfunction of your upkeep contract, which includes aStreamsLookuprevert custom error. TheStreamsLookuprevert enables your upkeep to fetch a report from Data Streams. If the report is fetched successfully, thecheckCallbackfunction is evaluated offchain. Otherwise, thecheckErrorHandlerfunction is evaluated offchain to determine

what Automation should do next.

In this example, thecheckErrorHandleris set to always returntrueforupkeepNeeded. This implies that the upkeep is always triggered, even if the report fetching fails. You can modify thecheckErrorHandlerfunction to handle errors offchain in a way that works for your specific use case. Read more aboutusing the StreamsLookup error handler.