Optimizing Gas Limit Settings in CCIP Messages

When constructing aCCIP message , it's crucial to set the gas limit accurately. The gas limit represents the maximum amount of gas consumed to execute theccipReceive 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 callsccipReceiveon the CCIP Receiver, which requires a manual re-execution with an increased gas limit. For more details about this scenario, read theManual Execution guide.
- Conversely, an excessively high gas limit leads to higher fees.

This tutorial shows you how to estimate the gas limit for theccipReceivefunction using various methods. You will learn how to use a CCIP Receiver where the gas consumption of theccipReceivefunction 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: UsingHardhat andFoundry 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.
2. 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.
3. Offchain Methods: Estimating gas with an offchain Web3 provider or tools likeTenderly 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 theccipReceivefunction, ensuring cost-effective CCIP transactions.

Before you begin, open a terminal, clone thesmart-contract-examples repository , and navigate to thesmart-contract-examples/ccip/estimate-gasdirectory.

gitclone https://github.com/smartcontractkit/smart-contract-examples.git&&\cdsmart-contract-examples/ccip/estimate-gas

Examine the code

The source code for the CCIP Sender and Receiver is located in thecontractsdirectory for Hardhat projects and in thesrcdirectory for Foundry projects. The code includes detailed comments for clarity and is designed to ensure self-explanatory functionality. This section focuses on the_ccipReceivefunction:

function_ccipReceive(Client.Any2EVMMessagememoryany2EvmMessage)internaloverride{uint256iterations=abi.decode(any2EvmMessage.data,
(uint256));uint256result=iterations;uint256maxIterations=iterations%100;for(uint256i=0;i<maxIterations;i++)
{result+=i;}emitMessageReceived(any2EvmMessage.messageId,any2EvmMessage.sourceChainSelector,abi.decode(any2EvmMessage.sender,(address)),iterations,maxIterations,result);}
The_ccipReceivefunction operates as follows:

1. Input Processing:The function accepts aClient.Any2EVMMessage. The first step involves decoding the number of iterations from the message's data using ABI decoding.
2. Logic Execution:It initializes theresultvariable with the number of iterations. The function calculatesmaxIterationsby 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.
3. Iteration:The function executes a loop from 0 tomaxIterations, simulating variable computational work based on the input data. This variability directly influences gas consumption.
4. Event Emission:Finally, an eventMessageReceivedis emitted.

This code shows how gas consumption for the_ccipReceivefunction can fluctuate in response to the input data, highlighting the necessity for thorough testing under different scenarios to determine the correctgasLimit.

Gas estimation in a local environment

To facilitate testing within a local environment, you will use theMockCCIPRouter 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 theMockCCIPRoutercontract is its ability to emit aMsgExecutedevent:

eventMsgExecuted(boolsuccess,bytesretData,uint256gassed)) This event reports the amount of gas consumed by theccipReceivefunction.

Foundry

Prerequisites

1. In your terminal, change to thefoundrydirectory:

cdfoundry 2. Ensure Foundry isinstalled . 3. Check the Foundry version:

forge--versionThe output should be similar to the following:

forge 0.2.0 (545cd0b 2024-03-14T00:20:00.210934000Z)You need version 0.2.0 or above. Runfoundryupto update Foundry if necessary. 4. Build your project:

forge buildThe output should be similar to:

[⠊] Compiling... [⠢] Compiling 52 files with 0.8.19 [⠆] Solc 0.8.19 finished in 2.55s Compiler run successful!

Estimate gas

Located in thetestdirectory, theSendReceive.t.soltest file assesses the gas consumption of theccipReceivefunction. This file features a test case that sends a CCIP message to theMockCCIPRoutercontract, which triggers theMsgExecutedevent. This event provides insights into the gas requirements of theccipReceivefunction 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 runs0iteration to determine the baseline gas consumption, representing the least amount of gas required.
- Average gas consumption:This scenario runs50iterations to estimate the gas consumption under average operational conditions.
- Peak gas consumption:This scenario runs99iterations to estimate the peak gas consumption, marking the upper limit of gas usage.

To run the test, execute the following command:

forgetest-vv--isolate Output example:

[⠊] Compiling... [⠘] Compiling 52 files with 0.8.19 [⠃] Solc 0.8.19 finished in 2.72s Compiler run successful!

Ran 3 tests for test/SendReceive.t.sol:SenderReceiverTest [PASS] test_SendReceiveAverage() (gas: 125166) Logs: Number of iterations 50 - Gas used: 14740

[PASS] test_SendReceiveMax() (gas: 134501) Logs: Number of iterations 99 - Gas used: 24099

[PASS] test_SendReceiveMin() (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:

ScenarioNumber of iterationsGas usedBaseline gas consumption05190Average gas consumption5014740Peak gas consumption9924099 The output demonstrates that gas consumption increases with the number of iterations, peaking when the iteration count reaches99. 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 thehardhatdirectory:

cd../hardhat 2. Install the dependencies:

npminstall 3. Set the password to encrypt your environment variables using the following command:

npx env-enc set-pw 4. Set the following environment variables to deploy contracts on testnets:

- PRIVATE_KEY: The private key for your testnet wallet. If you use MetaMask, follow the instructions toExport a Private Key .Note:Your private key is needed to sign any transactions you make such as making requests.
- ETHEREUM_SEPOLIA_RPC_URL: The RPC URL for Ethereum Sepolia testnet. You can sign up for a personal endpoint fromAlchemy ,Infura , or another node provider service.
- AVALANCHE_FUJI_RPC_URL: The RPC URL for Avalanche Fuji testnet. You can sign up for a personal endpoint fromInfura or another node provider service.

- ETHERSCAN_API_KEY: An Ethereum explorer API key, used to verify your contract. Followthis guide to get one from Etherscan.

Input these variables using the following command:

npx env-encset 5. Compile the contracts:

npx hardhat compileThe output should be similar to:

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 thetestdirectory, theSend-Receive.tstest file is designed to evaluate the gas usage of theccipReceivefunction. 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 theMockCCIPRoutercontract, triggering theMsgExecutedevent. This event provides insights into the gas requirements of theccipReceivefunction by detailing the amount of gas used.

To run the test, execute the following command:

npx hardhattest Example of the output:

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:

ScenarioNumber of iterationsGas usedBaseline gas consumption05168Average gas consumption5014718Peak gas consumption9924077 The output demonstrates that gas consumption increases with the number of iterations, peaking when the iteration count reaches99.

Compare the results from Foundry and Hardhat

This table summarizes the gas usage for different iterations from both Foundry and Hardhat:

ScenarioNumber of iterationsGas used (Foundry)Gas used (Hardhat)Baseline gas consumption051905168Average gas consumption501474014718Peak gas consumption992409924077 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 theccipReceivefunction 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.

EstimateccipReceivegas usage locally in your Foundry project

To estimate the gas usage of theccipReceivefunction within your own Foundry project, follow these steps:

1. Create a testing file in thetestdirectory of your project and import theMockCCIPRouter contract:

import{MockCCIPRouter}from"@chainlink/contracts-ccip/src/v0.8/ccip/test/mocks/MockRouter.sol";Note:TheMockCCIPRouterreceives the CCIP message from your CCIP Sender, calls theccipReceivefunction on your CCIP Receiver, and emits theMsgExecutedevent with the gas used. 2. Inside thesetUpfunction, deploy theMockCCIPRoutercontract, and use its address to deploy your CCIP Sender and CCIP Receiver contracts. For more details, check thisexample . 3. In your test cases:

1. Before transmitting any CCIP messages, usevm.recordLogs()to start capturing events. For more details, check thisexample .
2. After sending the CCIP message, usevm.getRecordedLogs()to collect the recorded logs. For more details, check thisexample .
3. Parse the logs to find theMsgExecuted(bool,bytes,uint256)event and extract the gas used. For more details, check thisexample .

EstimateccipReceivegas usage locally in your Hardhat project

To estimate the gas usage of theccipReceivefunction within your own Hardhat project, follow these steps:

1. Create a Solidity file in thecontractsdirectory of your project and import theMockCCIPRouter contract:

import{MockCCIPRouter}from"@chainlink/contracts-ccip/src/v0.8/ccip/test/mocks/MockRouter.sol";Note:TheMockCCIPRouterreceives the CCIP message from your CCIP Sender, calls theccipReceivefunction on your CCIP Receiver, and emits theMsgExecutedevent with the gas used. 2. Create a testing file in your project'stestdirectory. 3. Inside thedeployFixturefunction, deploy theMockCCIPRoutercontract and use its address to deploy your CCIP Sender and CCIP Receiver contracts. For more details, check thisexample . 4. In your test cases:

1. Send the CCIP message to theMockCCIPRoutercontract. For more details, check thisexample .
2. Parse the logs to find theMsgExecuted(bool,bytes,uint256)event and extract the gas used. For more details, check thisexample .

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.
2. Send several CCIP messages with the same number of iterations used in your local testing. For this purpose, utilize thesendCCIPMessage.tsscript in thescripts/testingdirectory. 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:

ScenarioNumber of iterationsEstimated gas usage (Hardhat)Buffered gas limit (+10%)Baseline gas consumption051685685Average gas consumption501471816190Peak gas consumption992407726485 3. UseTenderly to monitor and confirm that the transactions execute successfully within the buffered gas limits. Subsequently, compare the actual gas usage of theccipReceivefunction 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:

npx hardhat run scripts/deployment/deploySender.ts--networkavalancheFuji 2. Deploy the CCIP Receiver on the Ethereum Sepolia testnet:

npx hardhat run scripts/deployment/deployReceiver.ts--networkethereumSepolia 3. Authorize the Sender to send messages to Ethereum Sepolia:

npx hardhat run scripts/configuration/allowlistingForSender.ts--networkavalancheFuji 4. Authorize the Receiver to receive messages from the Sender:

npx hardhat run scripts/configuration/allowlistingForReceiver.ts--networkethereumSepolia

Upon completion, you will find the CCIP Sender and Receiver contracts deployed and configured on their respective testnets. Contract addresses are available in thescripts/generatedData.jsonfile.

Send CCIP Messages

1. Send three CCIP messages with different numbers of iterations:

npx hardhat run scripts/testing/sendCCIPMessages.ts--networkavalancheFujiExample output:

$ 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 2. Open theCCIP explorer , search each message by its ID, and wait for each message to be successfully transmitted (Statusin the explorer:Success).

For the example above, here are the destination transaction hashes:

Message idEthereum Sepolia transaction hash0xf23b17366d69159ea7d502835c4178a1c1d1d6325edf3d91dca08f2c7a2900f70xf004eb6dab30b3cfb9d1d631c3f9832410b8d4b3179e65b85730563b67b1e6890x4b3a97f6ac959f67d769492ab3e0 Notethat 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. OpenTenderly and search for thedestination transaction hash .
2. Search for_callWithExactGasSafeReturnDatawith a payload containing yourmessageId(without0x).Example for0xf23b17366d69159ea7d502835c4178a1c1d1d6325edf3d91dca08f2c7a2900f7.
3. Below the payload with yourmessageId, you will find the call trace from the Router to your Receiver contractCall trace example .
4. Click on theDebuggertab and you'll get the gas details:

"gas":{ "gas_left":5685 "gas_used":5031 "total_gas_used":7994315 } 5. Note thegas_leftis equal to the limit that is set in thesendCCIPMessages.tsscript:5685. Thegas_usedis the actual gas used by the Receiver contract to process the message. 6. Repeating the same steps for the other two messages, we can summarize the output:

ScenarioNumber of iterationsEstimated gas usage (Hardhat)Buffered gas limit (+10%)Gas used on testnetBaseline gas consumption0516856855031Average gas consumption50147181619014581Peak gas consumption99240772648523940

Testing on testnets has confirmed that a gas limit of 26,485 is adequate for theccipReceivefunction 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 usingoffchain methods to estimate gas usage.

Offchain methods

This section guides you through estimating gas usage using two different offchain methods:

- A Web3 provider using theethers.jsestimateGas function.
- Tenderly simulation API . The Tenderly simulation API provides a more accurate result (Read thisblog 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 theccipReceivefunction. 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 theoffchaindirectory:

cd../offchain 2. Modify thedata.jsonfile to insert the deployed addresses of your Sender and Receiver contracts. 3. Install the dependencies:

npminstall 4. Set the password to encrypt your environment variables:

npx env-enc set-pw 5. Set up the following environment variables:

- ETHEREUM_SEPOLIA_RPC_URL: The RPC URL for Ethereum Sepolia testnet. You can sign up for a personal endpoint fromAlchemy ,Infura , or another node provider service.
- TENDERLY_ACCOUNT_SLUG: This is one part of your Tenderly API URL. You canfind this value in your Tenderly account.
- TENDERLY_PROJECT_SLUG: This is one part of your Tenderly API URL. You canfind this value in your Tenderly account.
- TENDERLY_ACCESS_KEY: If you don't already have one, you cangenerate a new access token .

Input these variables using the following command:

npx env-encset 6. GenerateTypechain typings for the Receiver contract:

npmrun generate-types

Introduction of the scripts

The scripts are located in thesrcdirectory. Each script is self-explanatory and includes comprehensive comments to explain its functionality and usage. There are three scripts:

- estimateGasProvider.ts: This script uses theeth_estimateGas Ethereum API to estimate the gas usage of theccipReceivefunction. It simulates sending three CCIP messages to the Receiver contract with a varying number of iterations and estimates the gas usage using theethers.jsestimateGas function.
- estimateGasTenderly.ts: This script leverages the TenderlysimulateAPI to estimate the gas usage of theccipReceivefunction. 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 theTenderlysimulateAPI .

- 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 theestimateGasProvider.tsscript, 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 theccipReceivefunction.

Estimate gas using a Web3 provider

Ethereum nodes implement theeth_estimateGas Ethereum API to predict the gas required for a transaction's successful execution. To estimate the gas usage of theccipReceivefunction, you can directly call theeth_estimateGasAPI via a Web3 provider or leverage a library like ethers.js, simplifying this interaction. This guide focuses on theethers.jsestimateGas function for gas estimation. To estimate the gas usage, execute the following command in your terminal:

npmrun estimate-gas-provider Example output:

$ npm run estimate-gas-provider

> 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 theccipReceivefunction using Tenderly, execute the following command:

npmrun estimate-gas-tenderly Example output:

$ npm run estimate-gas-tenderly

> 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:

ScenarioNumber of iterationsGas estimated (Web3 provider)Gas estimated (Tenderly)Baseline gas consumption053775031Average gas consumption501494614581Peak gas consumption992432423940 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 theccipReceivefunction 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 theccipReceivefunction, 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.
2. 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.
3. 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 theccipReceivefunction 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 theEstimate 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.