Create Flexible, Secure, and Low-Cost Smart Contracts

In this guide, you will learn how the flexibility ofChainlink 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 ofChainlink Automation . If you are new to Automation, complete the following guides first:

- Learn how todeploy solidity contracts using Remix and Metamask
- Learn how to makecompatible contracts
- Register Upkeep for a smart contract

Chainlink Automation is supported on severalnetworks .

ERC677 Link

- GetLINK 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 toonvert Chainlink tokens (LINK) to be ERC-677 compatible.

Problem: Onchain computation leads to high gas fees

In the guide forCreating Compatible Contracts , you deployed a basiccounter 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) arraybalanceswith each element of the array starting with a balance of 1000.
- Thewithdraw()function decreases the balance of one or more indexes in thebalancesarray. 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:* ThecheckUpkeep()function checks if the contract requires work to be done. If one array element has a balance of less thanLIMIT, the function returnsupkeepNeeded == true.
- TheperformUpkeep()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 thanLIMIT, decreases the contractliquidity, and increases every found element to equalLIMIT.

// SPDX-License-Identifier: MITpragmasolidity^0.8.7;import{AutomationCompatibleInterface}from"@chainlink/contracts/src/v0.8/automation/interfaces/AutomationCompatibleInterface.sol";/ * **@dev Example contract which perform all the computation in performUpkeep * @notice important to implement {AutomationCompatibleInterface}** *// * 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. */contractBalancerOnChainisAutomationCompatibleInterface{uint256publicconstantSIZE=1000;uint256publicconstantLIMIT=1000;uint256[SIZE]publicbalances;uint256publicliquidity=1000000;constructor(){/// On the initialization of the contract, all the elements have a balance equal to the limitfor(uint256i=0;i<SIZE;i++){balances[i]=LIMIT;}}/// @dev called to increase the liquidity of the contractfunctionaddLiquidity(uint256liq)public{liquidity+=liq;}/// @dev withdraw an amountfrom multiple elements of balances array. The elements are provided in indexesfunctionwithdraw(uint256amount,uint256[]memoryindexes)public{for(uint256i=0;i<indexes.length;i++){require(indexes[i]<SIZE,"Provided index out of bound");balances[indexes[i]]-=amount;}}/// @dev this method is called by the Automation Nodes to check if performUpkeep should be performedfunctioncheckUpkeep(bytescalldata/*checkData*/)externalviewoverridereturns(boolupkeepNeeded,bytesmemoryperformData){upkeepNeeded=false;for(uint256i=0;i<SIZE&&!upkeepNeeded;i++){if(balances[i]<LIMIT){/// if one element has a balance < LIMIT then rebalancing is neededupkeepNeeded=true;}}return(upkeepNeeded,"");}/// @dev this method is called by the Automation Nodes. it increases all elements which balances are lower than the LIMITfunctionperformUpkeep(bytescalldata/*performData*/)externaloverride{uint256increment;uint256_balance;for(uint256i=0;i<SIZE;i++){_balance=balances[i];// best practice: reverify the upkeep is neededif(_balance<LIMIT){// calculate the increment neededincrement=LIMIT-_balance;// decrease the contract liquidity accordinglyliquidity-=increment;// rebalance the elementbalances[i]=LIMIT;}}}}
Open in Remix What is Remix? Test this example using the following steps:

1. Deploy the contract using Remix on thesupported testnet of your choice.
2. 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. Pass100,[10,100,300,350,500,600,670,700,900]to the withdraw function:

You can also perform this step after registering the upkeep if you need to. 3. Register the upkeep for your contract as explainedhere . Because this example has high gas requirements, specify the maximum allowed gas limit of2,500,000. 4. After the registration is confirmed, Automation Nodes perform the upkeep. 5. Click the transaction hash to see the transaction details in Etherscan. You can find how much gas was used in the upkeep transaction.

In this example, theperformUpkeep()function used2,481,379gas. This example has two main issues:

- All computation is done inperformUpkeep(). 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 thenetwork , which preventsperformUpkeepfrom 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 thecheckUpkeep()function. This computationdoesn't consume any gasand supports multiple upkeeps for the same contract to do the work in parallel. The main difference between this new contract and the previous contract are:

- ThecheckUpkeep()function receivescheckData , which passes arbitrary bytes to the function. Pass alowerBoundand anupperBoundto scope the work to a sub-array ofbalances. This creates several upkeeps with different values ofcheckData. The function loops over the sub-array and looks for the indexes of the elements that require re-balancing and calculates the requiredincrements. Then, it returnsupkeepNeeded == trueandperformData, which is calculated by encodingindexesandincrements. Note thatcheckUpkeep()is a view function, so computation does not consume any gas.
- TheperformUpkeep()function takesperformData as a parameter and decodes it to fetch theindexesand theincrements.

// SPDX-License-Identifier: MITpragmasolidity^0.8.7;import{AutomationCompatibleInterface}from"@chainlink/contracts/src/v0.8/automation/interfaces/AutomationCompatibleInterface.sol";/ * **@dev Example contract which perform most of the computation in checkUpkeep * * @notice important to implement {AutomationCompatibleInterface}** *// * 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. */contractBalancerOffChainisAutomationCompatibleInterface{uint256publicconstantSIZE=1000;uint256publicconstantLIMIT=1000;uint256[SIZE]publicbalances;uint256publicliquidity=1000000;constructor(){/// On the initialization of the contract, all the elements have a balance equal to the limitfor(uint256i=0;i<SIZE;i++){balances[i]=LIMIT;}}/// @dev called to increase the liquidity of the contractfunctionaddLiquidity(uint256liq)public{liquidity+=liq;}/// @dev withdraw an amountfrom multiple elements of thebalances array. The elements are provided in indexesfunctionwithdraw(uint256amount,uint256[]memoryindexes)public{for(uint256i=0;i<indexes.length;i++){require(indexes[i]<SIZE,"Provided index out of bound");balances[indexes[i]]-=amount;}}/ @dev this method is called by the Chainlink Automation Nodes to check if performUpkeep must be done. Note that checkData is used to segment the computation to subarrays. * * @dev checkData is an encoded binary data and which contains the lower bound and upper bound on which to perform the computation * * @dev return upkeepNeededif rebalancing must be done and performData which contains an array of indexes that require rebalancing and their increments. This will be used in performUpkeep */functioncheckUpkeep(bytescalldatacheckData)externalviewoverridereturns(boolupkeepNeeded,bytesmemoryperformData){/// perform the computation to a subarray ofbalances. This opens the possibility of having several checkUpkeeps done at the same time(uint256lowerBound,uint256upperBound)=abi.decode(checkData,(uint256,uint256));require(upperBound<SIZE&&lowerBound<upperBound,"Lowerbound and Upperbound not correct");// first get number of elements requiring updatesuint256counter;for(uint256i=0;i<upperBound-lowerBound+1;i++){if(balances[lowerBound+i]<LIMIT){counter++;}}// initialize array of elements requiring increments as long as the incrementsuint256[]memoryindexes=newuint256;uint256[]memoryincrements=newuint256;upkeepNeeded=false;uint256indexCounter;for(uint256i=0;i<upperBound-lowerBound+1;i++){if(balances[lowerBound+i]<LIMIT){/// if one element has a balance < LIMIT then rebalancing is neededupkeepNeeded=true;// store the index which needs increment as long as the incrementindexes[indexCounter]=lowerBound+i;increments[indexCounter]=LIMIT-balances[lowerBound+i];indexCounter++;}}performData=abi.encode(indexes,increments);return(upkeepNeeded,performData);}/ @dev this method is called by the Automation Nodes. it increases all elements whose balances are lower than the LIMIT. Note that the elements are bounded by lowerBoundand upperBound * (provided by performData * * @dev performData is an encoded binary data which contains the lower bound and upper bound of the subarray on which to perform the computation. * it also contains the increments * * @dev return upkeepNeededif rebalancing must be done and performData which contains an array of increments. This will be used inperformUpkeep */functionperformUpkeep(bytescalldataperformData)externaloverride{(uint256[]memoryindexes,uint256[]memoryincrements)=abi.decode(performData,(uint256[],uint256[]));uint256_balance;uint256_liquidity=liquidity;for(uint256i=0;i<indexes.length;i++){_balance=balances[indexes[i]]+increments[i];_liquidity-=increments[i];balances[indexes[i]]=_balance;}liquidity=_liquidity;}}
Open in Remix What is Remix? Run this example to compare the gas fees:

1. Deploy the contract using Remix on thesupported testnet of your choice.
2. Withdraw 100 at 10,100,300,350,500,600,670,700,900. Pass100,[10,100,300,350,500,600,670,700,900]to the withdraw function the same way that you did for theprevious example .
3. Register three upkeeps for your contract as explainedhere . Because the Automation Nodes handle much of the computation offchain, a gas limit of 200,000 is sufficient. For each registration,

pass the followingcheckDatavalues to specify which balance indexes the registration will monitor.Note: You must remove any breaking line when copying the values.

Upkeep NameCheckData(base16)Remark: calculated usingabi.encode()balancerOffChainSubset10x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0upperBound: 332balancerOffChainSubset20x000000000000000000000000000000000000000000000000000000000000014d0000000000000000000000000000000000000000000000000000000000000029alowerBou
333upperBound: 666balancerOffChainSubset30x000000000000000000000000000000000000000000000000000000000000029b00000000000000000000000000000000000000000000000000000000000003e7lowerBou
667upperBound: 999 4. After the registration is confirmed, the three upkeeps run: 5. Click each transaction hash to see the details of each transaction in Etherscan. Find the gas used by each of the upkeep transactions:

In this example the total gas used by eachperformUpkeep()function was 133,464 + 133,488 + 133,488 =400,440. This is an improvement of about 84% compared to the previous example, which used2,481,379gas.

Conclusion

Using Chainlink Automation efficiently not only allows you to reduce the gas fees, but also keeps them within predictable limits. That's the reason whyseveral Defi protocols outsource their maintenance tasks to Chainlink Automation.