

Create Flexible, Secure, and Low-Cost Smart Contracts

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](#).

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](#).

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-balance 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`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^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.

contract BalancerOnChain is AutomationCompatibleInterface {
    uint256 public constant SIZE = 1000;
    uint256 public constant LIMIT = 1000;
    uint256[SIZE] public balances;
    uint256 public liquidity = 1000000;

    constructor() {
        // On the initialization of the contract, all the elements have a balance equal to the limit
        for(uint256 i = 0; i < SIZE; i++) {
            balances[i] = LIMIT;
        }
        // @dev called to increase the liquidity of the contract
        function addLiquidity(uint256 liq) public {
            liquidity += liq;
        }
        // @dev withdraw an amount from multiple elements of balances array. The elements are provided in indexes
        function withdraw(uint256 amount, uint256[] memory indexes) public {
            for(uint256 i = 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 performed
        function checkUpkeep(bytes calldata) checkData {
            // external view override returns (bool upkeepNeeded, bytes memory performData)
            upkeepNeeded = false;
            for(uint256 i = 0; i < SIZE; i++) {
                if(balances[i] < LIMIT) {
                    // if one element has a balance < LIMIT then rebalancing is needed
                    upkeepNeeded = true;
                }
            }
            return (upkeepNeeded, "");
        }
        // @dev this method is called by the Automation Nodes. it increases all elements which balances are lower than the LIMIT
        function performUpkeep(bytes calldata) performData {
            // external override
            uint256 increment;
            uint256 _balance;
            for(uint256 i = 0; i < SIZE; i++) {
                _balance = balances[i];
                // best practice: reverify the upkeep is needed
                if(_balance < LIMIT) {
                    // calculate the increment needed
                    increment = LIMIT - _balance;
                    // decrease the contract liquidity accordingly
                    liquidity -= increment;
                    // rebalance the element
                    balances[i] = LIMIT;
                }
            }
        }
    }
}
```

[Open in Remix](#) [What is Remix?](#) Test this example using the following steps:

- Deploy the contract using Remix on the [supported testnet](#) of your choice.
- 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:

You can also perform this step after registering the upkeep if you need to. 3. 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. 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, 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 upkeeps 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 `lowerBound` and `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 `increments`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^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.

contract BalancerOffChain is AutomationCompatibleInterface {
    uint256 public constant SIZE = 1000;
    uint256 public constant LIMIT = 1000;
    uint256[SIZE] public balances;
    uint256 public liquidity = 1000000;

    constructor() {
        // On the initialization of the contract, all the elements have a balance equal to the limit
        for(uint256 i = 0; i < SIZE; i++) {
            balances[i] = LIMIT;
        }
        // @dev called to increase the liquidity of the contract
        function addLiquidity(uint256 liq) public {
            liquidity += liq;
        }
        // @dev withdraw an amount from multiple elements of the balances array. The elements are provided in indexes
        function withdraw(uint256 amount, uint256[] memory indexes) public {
            for(uint256 i = 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 upkeepNeeded if rebalancing must be done and performData which contains an array of indexes that require rebalancing and their increments. This will be used in performUpkeep
        function checkUpkeep(bytes calldata checkData) external view override returns (bool upkeepNeeded, bytes memory performData) {
            // perform the computation to a subarray of balances. This opens the possibility of having several checkUpkeeps done at the same time
            (uint256 lowerBound, uint256 upperBound) = abi.decode(checkData, (uint256, uint256));
            require(upperBound < SIZE && lowerBound < upperBound, "Lowerbound and Upperbound not correct");
            // first get number of elements requiring updates
            uint256 counter;
            for(uint256 i = 0; i < upperBound - lowerBound + 1; i++) {
                if(balances[lowerBound + i] < LIMIT) {
                    counter++;
                }
            }
            // initialize array of elements requiring increments as long as the increment
            uint256[] memory indexes = new uint256[upkeepNeeded ? counter : 0];
            uint256[] memory increments = new uint256[upkeepNeeded ? counter : 0];
            for(uint256 i = 0; i < counter; i++) {
                // if one element has a balance < LIMIT then rebalancing is needed
                if(balances[lowerBound + i] < LIMIT) {
                    // store the index which needs increment as long as the increment
                    indexes[i] = lowerBound + i;
                    increments[i] = LIMIT - balances[lowerBound + i];
                }
            }
            // 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 lowerBound and 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 upkeepNeeded if rebalancing must be done and performData which contains an array of increments. This will be used in performUpkeep
        function performUpkeep(bytes calldata performData) external override {
            (uint256[] memory indexes, uint256[] memory increments) = abi.decode(performData, (uint256[], uint256[]));
            uint256 _balance;
            liquidity = liquidity;
            for(uint256 i = 0; i < indexes.length; i++) {
                _balance = balances[indexes[i]] + increments[i];
                liquidity -= increments[i];
                balances[indexes[i]] = _balance;
            }
        }
    }
}
```

[Open in Remix](#) [What is Remix?](#) Run this example to compare the gas fees:

- Deploy the contract using Remix on the [supported testnet](#) of your choice.
- 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](#).
- 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.

[illegible]

In this example the total gas used by each `performUpkeep()` function was $133,464 + 133,488 + 133,488 = 400,440$. This is an improvement of about 84% compared to the previous example, which used 2,481,379 gas.

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 why [several DeFi protocols](#) outsource their maintenance tasks to Chainlink Automation.