Optimising

the gas costs of your solidity smart contracts

can save more than 90% in transactions

to you and your users, making your protocol: more scalable, cheaper, and successful long term.

In this guide on the best Solidity gas optimization tips and techniques,

you will learn 11 advanced, real-world, and tested strategies taught by top-notch web3 developers to reduce the gas costs of your smart contracts.

Keep in mind, that the examples in this guide come from really simple contracts, are for demonstration purposes only, and in most cases, only take into consideration runtime Gas costs, as deployment costs can greatly vary based on the size of the smart contract.

In real-life scenarios, we strongly suggest each smart contract should go through a complete and in-depth

[auditing process

](https://www.cyfrin.io/blog/what-is-a-smart-contract-audit).

For all the examples and tests in this article, you can refer to theGithub gas optimization tips repository.

Before getting started with this web3 development guide, let's quickly refresh why gas optimization is important!

The importance of Solidity gas optimization

Gas optimization is crucial for developers, users, and the long-term success of projects and protocols. Efficiently optimizing the gas of your smart contracts,

will make your protocol more cost-effective,

and scalable

while reducing security risks such as DoS.

Gas-efficient contracts improve the usability of your product and your UX, enabling faster and cheaper transactions even under congested network conditions.

Simply put, optimising the gas costs makes Solidity smart contracts, protocols, and projects:

- Cost-effective
- Efficient
- Usable

While improving adoption and giving a competitive advantage to the more efficient ones.

On top of this, improving smart contracts' code helps uncover potential vulnerabilities

, making your protocol and its users, more secure.

Note

: This guide doesn't substitute in any way a thorough security review done bytop smart contracts auditing firms in web3

In summary, gas optimization should be a key focus during development, as it isn't just a nice-to-have

but a must-have

for the long-term success and security of a smart contract.

Without further ado, let's delve into the most effective techniques for optimizing gas usage.

Disclaimer

: all the tests in this guide are done using Foundryand the following setup:

- Solidity version

: ^0.8.13;

- Local Blockchain Node:

Anvil

- Command used:

forge test

- Optimization runs:

100

Every test has been run 100 times, and all the results in this guide are the average of results between all tests.

Solidity Gas Optimization Tips

1. Minimize on-chain data

As a developer, it is crucial to question the necessity of recording on a chain all user data, NFT game statistics, or any other extensive information Solidity smart contracts might handle.

By allocating less storage to store variables, you can significantly reduce the gas consumption of your smart contracts.

One effective approach to achieve this is: using events to store data off-chain instead of storing data directly on-chain

.

Using events will inevitably increase the gas cost for each transaction

due to the extra emit function. However, the savings gained from not storing the information on-chain outweigh this cost.

Take this Contract into consideration, which pushes all the data to a struct

each time the vote

the function is executed.

Contract Link.

Testing the vote

function using Foundry over 100 times we get these results:

On the other hand, we have another smart contract that is not going to store the information on-chain but instead just emits an event each time the vote function is called.

Contract Link.

As you can see, just by minimizing on-chain data, we saved an outstanding 90.34%

gas on average.

If you want to get your off-chain stored data you can then use something likeChainlink functions.

Solidity minimizing on-chain data Test:

Gas usage before optimization:

23,553

Gas usage after optimization:

2,274

Gas usage average reduction: 90.34%

Test Url on GitHub.

2. Use Mappings over Arrays

In Solidity, mappings serve as an excellent tool for establishing relationships between multiple pieces of information. When it comes to representing lists of data, Solidity offers two data types: arrays and mappings.

Arrays

store a collection of items, each assigned to a specific index. On the other hand, mappings

are key-value data structures that provide direct access to data through unique keys.

While arrays might be useful to store vectors and similar data, mappings are generally recommended to use whenever possible,

especially when used to store items that need to be retrieved on demand such as names or wallets and balances.

To understand why, we need to remember that even if read functions in Solidity smart contracts are free when called from an EOA

, they still incur costs when called as part transactions

, and are charged based on the gas consumed. If to retrieve a value, we need to loop through every item of an array, we'll need to pay for each gas consumed by the related

[opcodes

](https://ethereum.org/en/developers/docs/evm/opcodes/).

To illustrate this concept, here's an example showcasing the use of arrays and their equivalent mappings in Solidity:

Contract link.

In the above example, we are using an array to store the addresses of users and their corresponding balances.

When we need to retrieve the user's balance, we'll have to loop through each item, look if the userAddress matches the _userAddress argument, and if it matches, return the balance.

Messy, right?

Instead, we can use a mapping to directly access the balance of a particular user without having to iterate through all the elements in the array:

Contract Link.

By replacing the array with a mapping, we save on gas costs as we no longer need to loop through all the elements to fetch the data we need.

In this test, just by substituting the array with a mapping, we have optimized our gas by 93% on average

.

And this also applies to the retrieval function which has an outstanding saving of 89% in gas cost.

Solidity mappings over arrays test:

Gas usage before optimization: 30,586

Gas usage after optimization: 3,081

Gas saved: 89%

Test link on Github.

3. Use Constant and Immutable to reduce smart contracts gas costs

Another tip when optimising the gas costs of your Solidity smart contracts, is using constants and immutable variables as, unlike other variables, do not consume storage space within the Ethereum Virtual Machine (EVM).

Their values are instead compiled directly into the smart contract bytecode

, resulting in reduced gas costs associated with storage operations

.

When declaring variables as "immutable" or constant in Solidity, values are assigned exclusively during contract creation and become read-only

thereafter, this makes immutable and constant variables more cost-effective to access than regular state variables - as they reduce the gas required for SLOAD operations.

Take into consideration this example:

Contract link

As you can see here, we're declaring our variables maxSupply

and owner

without using the constant or immutable keywords. Running our test 100 times, we'll get an average gas cost of 112,222 units:

As maxSupply and owner are known values and aren't planned to be changed

, we can declare a maximum supply and an owner for our smart contract that does not consume any storage space.

Let's add the constant

and immutable

keywords, slightly changing the declaration:

Contract link

By simply adding the immutable

and const keywords

to our variables in our Solidity smart contract - we now have optimized the average gas spent by a significant 35.89%

.

Solidity constant vs immutable Test:

Gas usage before optimization: 112,222

Gas usage after optimization: 71,940

Gas saved: 35.89%

Test link on Github.

4. Optimise Unused Variables

Optimizing the variables in a Solidity smart contract is one of those Solidity gas optimization tips that sound obvious. The truth is that it happens a lot that not useful variables are kept in the execution of smart contracts, ending up in avoidable gas costs

.

Take a look at the following example of a bad use of variables

:

Contract link

In this contract, unusedVariable

is declared and manipulated in the calculate

function, but it is never used anywhere else, neither in the same function nor elsewhere in the contract.

Let’s see how much gas that unused variable is costing us

:

Let’s now optimize our contract, by removing the unusedVariable:

Contract link

As you can see, just by removing one single unused variable

, in our smart contract, we’ve been able to reduce our smart contract gas costs by an average of 18%

Solidity optimize unused variables test:

Gas usage before optimization: 32,513

Gas usage after optimization: 27,429

Gas saved: 18%

Test link on

[Github

](https://github.com/Cyfrin/gas-optimization-tips/blob/main/test/VariableEfficiencyTest.t.sol).

5. Solidity Gas refund deleting unused variables

Deleting unused variables doesn’t mean “deleting” them, as it would cause all sorts of issues to pointers in memory - it's more like assigning a default

value back to a variable, that when done, grants you a 15,000 units Gas refund.

For example, issuing a delete

to a uint

variable simply sets the variable's value to 0

.

Let’s take a look at a really simple example where we have a variable call data

which can store a uint

:

Contract link.

In this case, we’re not deleting our “data” variable once the function ends, paying a total of 100,300 gas units on average - just to assign that variable to data.

Now let’s see what happens when we use the

[delete

](https://docs.soliditylang.org/en/develop/types.html#data-location)keyword

:

Contract link.

Just by deleting our “data” variable, that is: setting its value back to 0,

we’re saving 19% Gas on average!

Solidity deletes unused variables test:

Gas usage before optimization: 100,300

Gas usage after optimization: 80,406

Gas saved: 19%

Test link on Github.

6. Use Fixed sized Arrays over Dynamics to reduce your smart contract gas costs

As mentioned earlier, to optimize the gas of your Solidity smart contracts you should use mappings whenever possible

.

However, if you find yourself in the need of using arrays in your contracts, it's better to do it by trying to use fixed-sized arrays and avoid dynamically sized ones

, as they can grow indefinitely, resulting in higher gas costs.

Simply put, statically sized arrays have known lengths, hence when the EVM needs to store one, it doesn’t need to keep the length of the array readily available in the storage

:

On the other hand, dynamically sized arrays can grow in size, hence the EVM needs to keep track of and update their length every time a new item is added:

Let’s take a look at the following code where we declare a dynamically sized array and update it through the updateArray function:

Contract link

Notice that we use a require

statement to ensure that the supplied index isn't out of the range of our fixed-size array.

Running our test 100 times will result in 12,541 gas units spent on average

.

Now, let’s modify our array to be of fixed size 5:

Contract link

In this example, we define a fixed-size array of length 5 of type uint256

. The updateArray

function, the same as before, allows us to update the value at a specific index of our array.

The EVM will now know that the state variable fixedArray

is of size 5, and will allocate 5 slots to it, without having to store its length in storage.

Running the same test 100 times, just by using a fixed array instead of a dynamic one

, we have saved 17.99% of gas costs

.

Optimize unused variables test:

Gas usage before optimization: 12,541

Gas usage after optimization: 10,284

Gas saved: 17.99%

Test link on Github.

7. Avoid using lower than 256bit variables

Using uint8

instead of uint256

in Solidity can be less efficient and potentially more costly in certain contexts, primarily due to the way the Ethereum Virtual Machine (EVM) operates.

The EVM operates with a word size of 256 bits.

This means that operations on 256-bit integers (uint256

) are generally the most efficient, as they align with the EVM's native word size. When you use smaller integers like uint8

, Solidity often needs to perform additional operations to align these smaller types with the EVM's 256-bit word size. This can result in more complex and less efficient code.

While using smaller types like uint8

can be beneficial for optimizing storage (since multiple uint8

variables can be packed into a single 256-bit storage slot

), this benefit is typically seen only in storage and not in memory or stack operations.

On top of it, for computations, the conversion to and from uint256

can negate the storage savings.

In summary, while using uint8

may seem like a good way to save space and potentially reduce costs, but in practice, it can lead to inefficiencies and increased gas costs due to the EVM's optimization for 256-bit operations.

You can create transactions that invoke a function f(uint8 x)

with a raw byte argument of 0xff000001

and 0x00000001

. Both are supplied to the contract and will appear as the number 1

to x

. However, msg.data

will differ in each case. Therefore, if your code implements things like keccak256(msg.data)

running any logic, you will get different results.

8. Pack smaller than 256-bit variables together

As we said before, using lower than 256-bit int or uint variables is generally considered less efficient than 256 variables, but there are situations where you’re forced to use smaller types, such as when using booleans

that weigh 1 byte, or 8-bit.

In these cases, by declaring your state variables with the storage space in mind, Solidity will allow you to pack them, and store them all in the same slot.

Note

: The benefit of packing variables is typically seen only in storage and not in memory or stack operations.

Let’s take into consideration the following example:

Contract link.

Considering what we've said before, that each storage slot in Solidity has a space of 32 bytes equal to 256-bit

, in the example above we'll have to use 3 storage slots to store our variables

:

- 1 to store our boolean "a" (1 byte)
- 1 to store our uint256 "b"(32 bytes)
- 1 to store our boolean "c" (1 byte).

Each storage slot used incurs a gas cost, hence we're spending 3 times that cost.

Given that the combined size of the two boolean variables is 16 bits, which is 240 bits less than a single storage slot's capacity, we can instruct Solidity to store variables "a" and "c" in the same slot

, also said: "we can pack them".

Packing the variables together allows you to lower your smart contract deployment gas costs by reducing the number of slots required to store state variables.

We can pack these variables together by re-ordering their declarations as follows:

Contract link.

Solidity will pack the two boolean variables

together in the same slot as they weigh less than 256-bit

or 32 bytes.

That said, keep in mind that we're still potentially wasting storage space. The EVM operates on 256-bit words

and will have to perform operations to normalize smaller-sized words. This might offset any potential gas savings.

Running our test over 100 iterations we get an average of 13% optimization

.

Variables packing test:

Gas usage before optimization: 1,678

Gas usage after optimization: 1,447

Gas saved: 13%

Test link on Github.

9. Use External Visibility Modifier

In Solidity, choosing the most appropriate visibility

for functions can be an effective measure to optimize you smart contract gas consumption.

Specifically, using the external

visibility modifier can be more gas-efficient than public

.

The reason has to do with how public functions handle arguments, and how the data is passed to these functions.

External

functions can read from calldata

,

a read-only, temporary area in the EVM storing function call parameters. Using calldata

is more gas-efficient for external calls because it avoids copying data from the transaction data to memory

.

On the other hand, functions declared as public

can be called both internally (from within the contract) and externally

. When called externally, they behave similarly to external

functions, with parameters passed in the transaction data. However, when called internally, the parameters are passed in memory, not in calldata

.

Simply put, since public

functions need to support both internal and external calls

, they cannot be restricted to only accessing calldata

.

Consider the following Solidity contract:

Contract link.

This function calculates the sum of an array of numbers. Because the function is public, it has to accept an array from memory

which, if large, can be costly in terms of gas.

Now let's modify this function by making it external:

Contract link.

By changing the function to external

, we can now accept arrays from calldata

, making it more gas-efficient when dealing with large arrays.

This highlights the importance of properly using visibility modifiers in your Solidity smart contracts to optimize gas usage.

In this case, modifying your Solidity function modifiers leads to saving on average 0.3% Gas units each call.

Optimize unused variables test:

Gas usage before optimization: 495,234

Gas usage after optimization: 493,693

Gas saved: 0.3%

Test link on Github.

10. Enable Solidity Compiler Optimization

Solidity comes equipped with a compiler with easy-to-modify settings to optimize your codebase compiled code.

Consider the Solidity compiler as a wizard's spell book, and your intelligent manipulation of its options can create potions of optimization that significantly reduce gas usage

.

The --optimize

option is one such spell you can cast.

When enabled, it performs several hundred runs, streamlining your bytecode

and translating it into a leaner version that consumes less gas

.

The compiler can be adjusted to strike a balance between deployment cost and runtime cost.

For instance, using the --runs

command

lets you define the estimated number of executions

your contract will have.

- Higher number:

the compiler optimizes for cheaper gas costs during contract execution.

- Lower number:

the compiler optimizes for cheaper gas costs during contract deployment.

By using the --optimize

flag and specifying --runs=200

, we instruct the compiler to optimize the code to reduce gas consumption during contract executions by running the incrementCount

function around 200 times.

Make sure to adjust these settings based on your application's unique needs.

11. Extra Solidity Gas Optimization tip: use Assembly*

When you compile a Solidity smart contract, the compiler will transform it into bytecodes, a series of EVM (Ethereum Virtual Machine) opcodes.

By using assembly, you can write code that operates at a level closely aligned with opcodes.

While it may not be the easiest task to write code at such a low level, the advantage lies in the ability to manually optimize the opcodes

, thereby outperforming Solidity bytecode in certain scenarios.

This level of optimization allows for greater efficiency

and effectiveness

in contract execution.

In a simple example with two functions intended to add two numbers, one using plain solidity and the other one using assembly we have small differences but the assembly one is still cheaper.

Now implementing assembly:

We want to make an honorific mention toHuff, which allows us to write assembly with prettier syntax.

Note

: Even if using Assembly might help you optimize the Gas cost of your smart contracts, it might also lead to insecure code. We strongly recommend having your contracts reviewed by

[smart contract security experts

](/blog/top-10-smart-contract-auditing-companies)before deploying

.

Conclusion

Optimizing gas usage in Solidity is essential for creating cost-effective,

high-performing, and sustainable Solidity smart contracts

. By implementing the Solidity gas optimization tips you've learned in this guide, you can reduce transaction costs, improve scalability, and enhance the overall efficiency of your contracts.