# 295.https://stackoverflow.com/questions/71752162/is-it-considered-good-practice-to-use-contract-functions-once-when-deployed

**T:**Is it considered good practice to use contract functions once when deployed?

**Q:**I have many getters and setters in my contract, for example:  
  
function setMintRate(uint256 \_mintRate) public onlyOwner { mintRate = \_mintRate;}  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
If I deploy my contract to mainnet Ethereum, is it considered good practice to call these functions from Remix after the contract is deployed in order to maintain/update my project? For example updating the price of an NFT mint. If not, what is the correct way of doing it?  
  
Thanks!

3 **Answer**

**A1:**As the previous answers state, in order to take advantage of the decentralized feature of blockchain, the smart contracts deployed on it should be trustless and de-centralized. For me, this means that right after deploying the contract, it should work autonomously and no single entity should be able to modify it.  
  
Enter governance: a way to allow participants of your dApp to vote for changes and proposals that would modify a smart contract of the dApp. There are amazing tutorials for it such as:  
  
 ● How to set up on-chain governance  
  
 ● How to Code an On-Chain DAO  
  
 ● COMP Governance Explained  
  
 ● How to create an NFT DAO  
  
To briefly explain it here, the governance system should have 4 components:  
  
 ● The modifiable smart contract: it is your current smart contract that has specific setter functions with the onlyOwner modifier. As of right now, the owner (a centralized entity) is the only one who can execute the setter function and modify the values of the smart contract.  
  
  
 ● The votes smart contract: it is a ERC-20 or a ERC-721 contract that inherits a Votes module that keeps tabs of how many tokens each address has in a specific time. This contract defines the voting power of each user/address (in a ERC-20 usual case it would depend on the amount of tokens the address has, and in a ERC-721 case it would be something like one-token one-vote)  
  
  
 ● The governor smart contract: it is the contract that would get proposals and determine the number of votes each proposal has. A user would send a proposal to be voted that would contain the transaction byte code to be executed in the modifiable contract. The governor contract would take care of the election process and later on would count how many votes the proposal got.  
  
  
 ● The timelock smart contract: this contract would need to be transferred the "ownership" of the modified contract after deployment. As the name states, it is an automonous contract that would wait a time interval to execute the winning proposal so users who not agree with the new proposal have enough time to leave the dApp.  
  
  
I hope this summary helped but I highly recommend to read the amazing tutorials attached above.

**C1:**Solid, Thanks for this, I've been looking for the hardhat version of the dao template for a while

**A2:**The best option is to set up contract params once, through constructor call, like:  
  
constructor(uint256 \_mintRate) { mintRate = \_mintRate;}  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Otherwise, when you give backdoor to manipulate contract, e.g mint price, by single user(onlyOwner) then it'll become trustless.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
However, when you introduce governance to your system, then community can make decision if they want to make any changes in system.

**A3:**You would want to set the initial state of your contract in your constructor, during deployment.  
  
However, the needs may change in the future and it is perfectly fine to use setters, but it is good to keep some practices in mind.  
  
Let me share a couple of anectodes with you.  
  
 ● Try avoiding scammy stuff.Add governance, access control (ownership,etc) to your setter functions. Else, it might seem like you'll walk away with anything in the contract  
  
  
 ● Make sure your setter functions as simple as possible. Sometimes, you might feel the need to create complex setters. For example, merging multiple setters to cut deployment costs(?). Having complex setters can lead to vulnerabilities since it is easy to oversee details.  
  
  
 ● Try having high test coverage in general, but at places you change your contracts state (like setters), go for as high as possible.

**C1:**Thanks for your answer. What is the difference between access control and ownership ? Isn't it the same?

**C2:**I can see how that was misleading. We can say that access control is more of a general term and ownership is a way to satisfy it