J Jeffrey Scholz Jan 3 7 min read

The Architecture of the Compound V3 Smart Contract

Introduction and prerequisites

Compound is one of the most significant lending protocols in DeFi, having inspired the design of nearly every lending protocol on several blockchains. This article explains the smart contract architecture of V3.

Since Compound is a lending protocol, we assume the reader is familiar with how interest rates work in DeFi and are familiar with liquidations and collateral in the context of DeFi loans.

Unlike Compound V2, each instance of Compound only lends one asset. That is, you can only borrow USDC from the USDC market and can only borrow ETH from the ETH market -- you cannot borrow anything else. To borrow these assets, you need to supply collateral according to the collateralization ratio specified by the market (and that ratio is set by governance).

The borrowable asset is called the **base asset** in Compound V3. Because USDC is the most popular asset to borrow and lend, we will use USDC as the base asset in our running examples. The smart contract that handles the core logic of lending and borrowing is referred to as **Comet** in their literature and codebase. Compound has instances on Ethereum mainnet and on various L2s: Polygon, Base, and Arbitrum at this time of writing. The list of available Compound V3 markets is here.

This article gives a high level overview of how to use this smart contract and the architecture of the Solidity files.

Part 1: Using Compound

There are three major actions that can be taken with Compound V3:

1. lending the base asset
2. supplying collateral and borrowing the base asset
3. liquidating under-collateralized loans.

Lending USDC

The following video shows the actions for lending USDC on the Polygon network (USDC on Polygon market link).

0:00 / 0:29

People who participate in the protocol are awarded COMP token rewards. The account above has accrued 0.0004 COMP rewards so far as indicated by the pink arrow. The rewards have not been claimed yet. Additionally, it has accrued over 6 cents in interest so far (4,602.0649 - 4,602.00 = 0.0649).

The mismatch between the prices in the green box and the yellow box is because USDC is not always exactly $1.00. The chart below shows Chainlink's recent oracle prices of USDC / USD on Polygon and the reader can see it is not always exactly one dollar.

Borrowing USDC

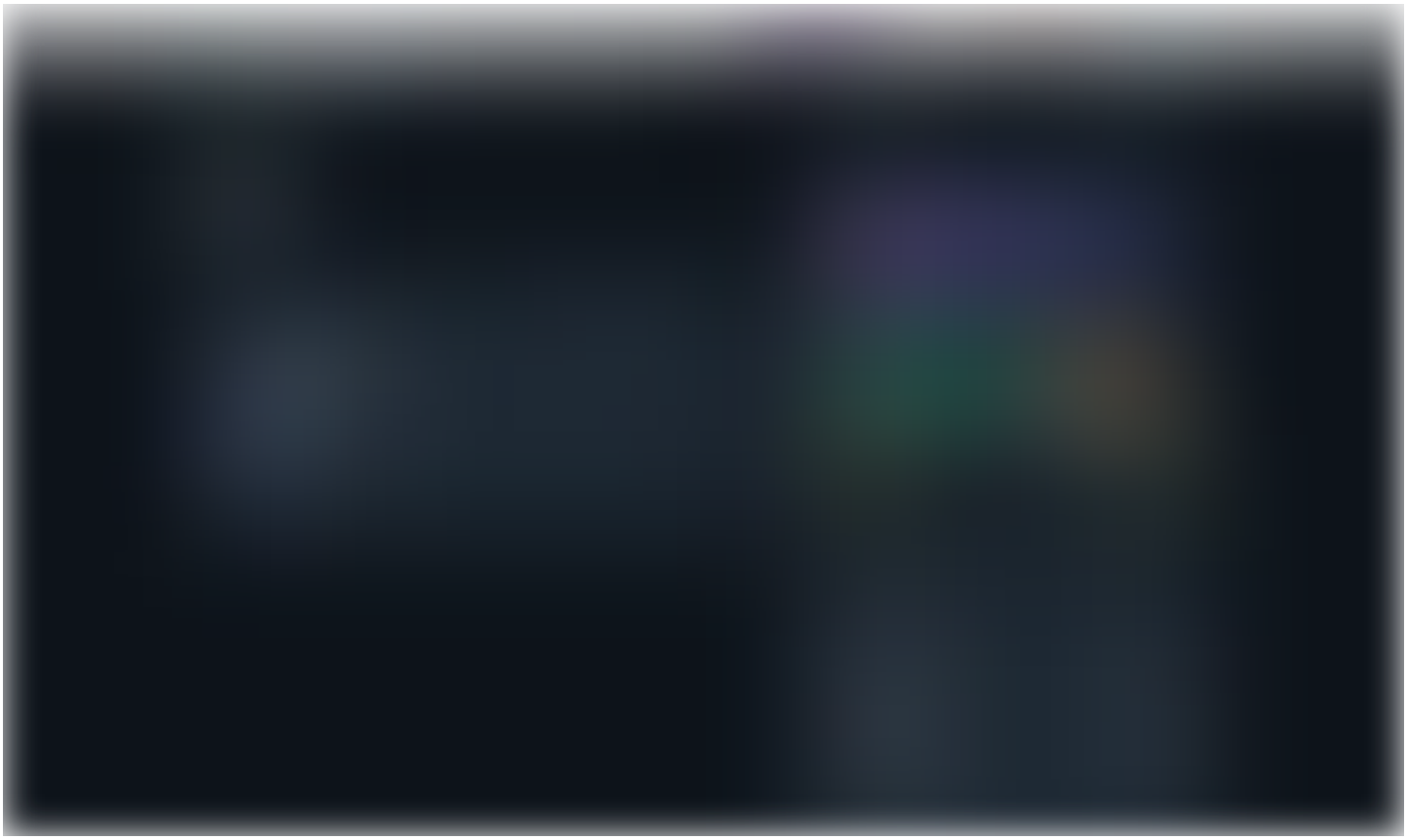
The following video shows the procedure for borrowing. This time the account supplied 0.06 ETH ($110) as collateral and borrowed $100 of USDC. This was done on the BASE L2 lending market.

0:00 / 0:37

Checking the browser wallet, the account now has 100 Base USDC in it. USDCbC is USDC bridged to Base.

The following video shows the user paying back the USDC (before interest significantly accumulated) and withdrawing the ETH collateral.

Understanding net borrow and net lending rates



In the screenshot above, note that the Net Supply APR (lending interest rate) is higher than the Net Borrow APR (yellow circles). This normally isn't possible because lenders cannot earn more than what borrowers are paying. Compound is including the value of COMP rewards into the calculation. Specifically, the borrowers currently pay 6.71% interest (top red circle) but earn 2.58% interest in COMP (top blue circle). 6.71 - 2.58 = 4.13%, which is the Net Borrow APR.

Similarly, lenders currently earn 6.47% from lending USDC (lower red circle), but also earn an additional 4.63% from the value of COMP rewards (lower blue circle). The sum of those two is 11.10%.

Since borrowers are paying 6.71% interest in USDC and lenders are collecting 6.47% interest, 0.24% interest in USDC is going to the protocol.

COMP rewards rates change subject to governance votes.

Part 2: Navigating the codebase

Compound V3 has 4,304 lines of Solidity code, not counting comments or blank spaces.

Compound V3 architecture from 10,000 feet

Below we have screenshotted the Github repo of Compound V3.

- **All of the files highlighted in green hold the core borrowing and lending functionality. The inheritance relationship between them will be shown later. Comet is the primary lending and borrowing smart contract of Compound V3.**
- **All the files highlighted in blue form the smart contract that deploys new Comet instances during an upgrade. Again, the inheritance relationship between them will be shown later.**
- **The pink highlighted file is the contract that distributes rewards**

The following diagram summarizes the deployed smart contracts that make up Compound V3. This is only a high level overview, a more detailed one will be given later. Note that the color coding matches the highlights above. Specifically, the primary lending and borrowing contract (Comet) is green, the contracts related to deploying new Comet instances are blue, and the reward contract is pink.

Most users will interact with Compound V3 through the comet proxy (not shown in Github) or with the rewards contract to earn COMP for participating as a lender or borrower. All of the user facing logic is in the comet contract where the comet proxy delegates its functionality to.

The configuration and factory contracts are to deploy new comet instances when governance votes for an upgrade.

The smart contracts with an asterisk (*) have several ancestor contracts. In the next section, we show the inheritance chain for Comet.

Compound V3 has an unusual method of updating parameters based on governance

Interest rate models can be changed by governance if crypto-economic conditions change, and so can the liquidation factors. Compound stores all of this information in immutable variables, not storage. To alter these values, a new Comet instance must be deployed, and the proxy is updated to the new implementation contract.

This may seem like an odd design choice, but it has several advantages:

- **immutable variables are much more gas efficient than storage variables**

- **The core contracts don't need to clutter themselves with setter functions**

We will examine the lifecycle of a parameter change later.

Comet Inheritance

Comet's ancestors are shown in the diagram below. We will give a high level overview of each of the ancestor contracts in this section.

CometMath.sol (top left gray ellipse)

Comet math simply contains a bunch of functions for casting unsigned integers to unsigned integers of lower bit value and reverting if it would cause an overflow. For example, if we cast from uint256 to uint104, but the value of the uint256 is larger than what can be stored in uint104, it will revert. You'll see functions like safe64 scattered throughout the codebase. Hopefully these are easy enough to understand without further explanation. The file is small, so we show it in its entirety here:

CometStorage.sol (red ellipse)

Every single storage variable used by Comet is defined in CometStorage.sol and nowhere else. That is, no other contract in the Comet inheritance chain besides CometStorage has any storage variables.

CometCore.sol (second row gray ellipse)

CometCore.sol defines how Compound tracks and interprets the accumulation of interest for lenders and borrowers. It also defines some global constants. We will dive into this file more when we discuss Principal Value and Present Value.

CometMainInterface.sol (left blue ellipse)

As the name implies, CometMainInterface.sol is simply an interface file, except with the quirk that it is defined as an abstract contract. Since interfaces are self-explanatory, we omit discussion of this.

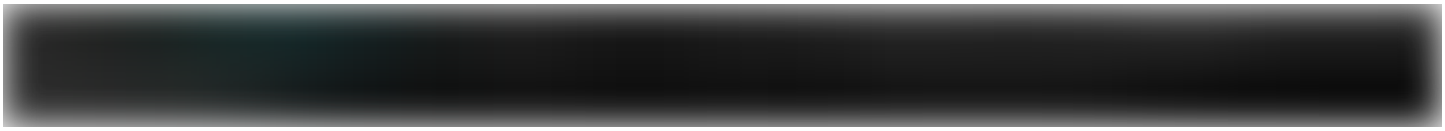
Comet.sol (green ellipse)

Comet.sol is the star of the show. It provides all the public facing functions for users to lend, borrow, repay, and liquidate loans.

CometExt is an extension to Comet via delegatecall

To avoid hitting the 24kb deployment limit, Comet offloads several extra functions to CometExt using the fallback-extension pattern. For example, the function name() is not in Comet.sol and thus cannot be seen on Etherscan.

However, if we call that function via Foundry cast, we can see the contract behaves as if it exists.



What happens is that function call hits the fallback function and delegates the call to CometExt, which has a function called name() in it.

It is important that CometExt respects the storage layout of Comet and extends it, without clashing. This is achieved by having CometExt mimic Comet's inheritance structure.

Remember, ***inheritance is just a semantic description — when contracts are deployed, the entire inheritance chain becomes one contract. It is not possible for a deployed contract to inherit from another deployed contract.***

A more verbose diagram showing the relationship between Comet and CometExt is shown here

Rewards Issuance

There is only one contract for handling non-interest reward issuance show in the pink box.

Lenders and borrowers are rewarded with COMP tokens for their participation in the ecosystem. The Comet contract keeps track of participation, but it does not handle reward issuance. The Reward contract reads the state of Comet and issues COMP tokens. The rate at which rewards are issued are parameters inside the Reward contract that are set by governance.

Lifecycle of a parameter update

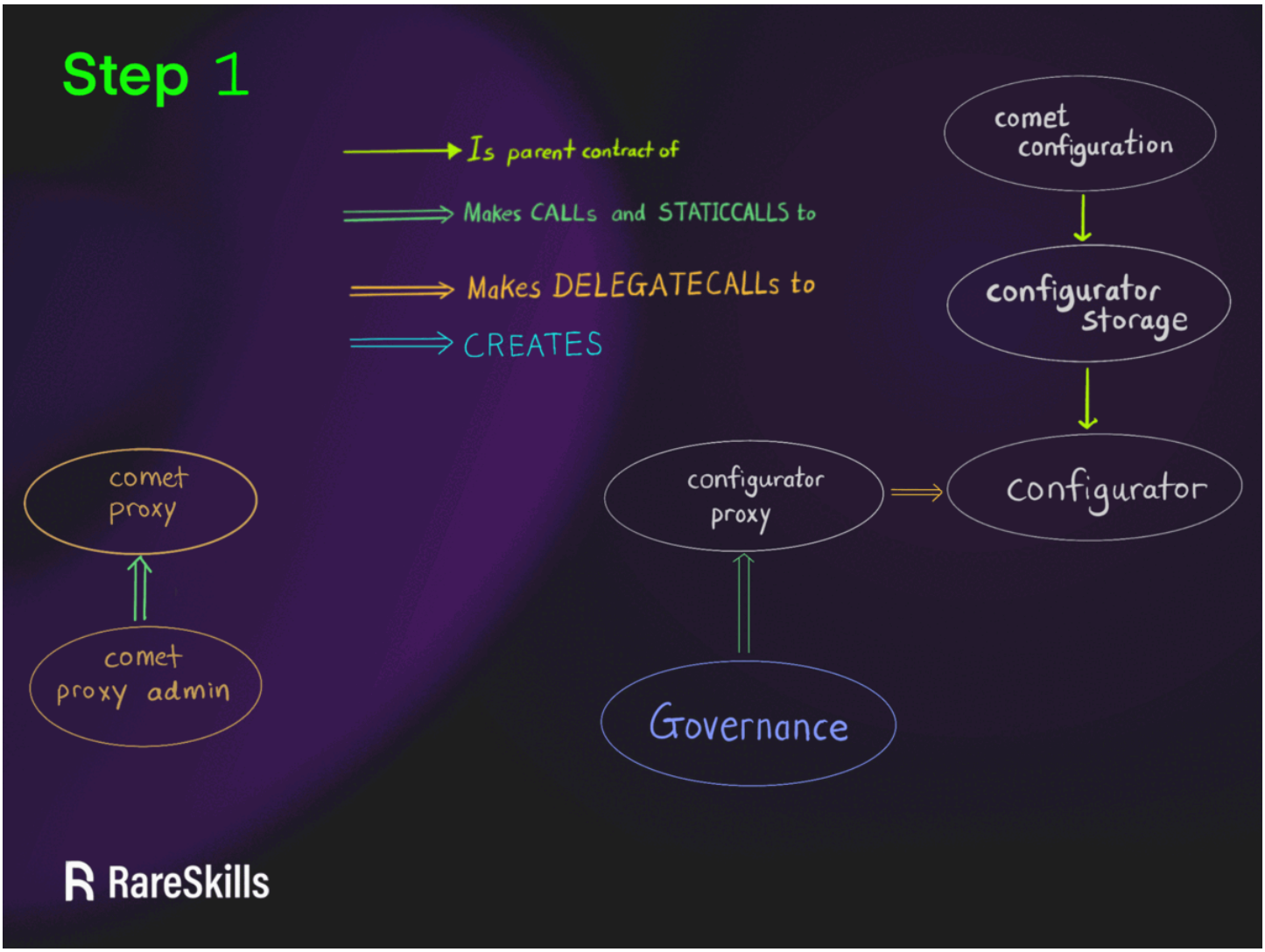
As mentioned earlier, Comet is parameterized by immutable variables. Changing these parameters requires updating the proxy to point to a new implementation deployed by the Configuration contracts.

In practice, changing the interest rate curves is very infrequent. The mainnet contract has only undergone three interest rate curve updates:

https://compound.finance/governance/proposals/162 (May 30, 2023)
https://compound.finance/governance/proposals/168 (July 17, 2023)
https://compound.finance/governance/proposals/201 (Dec 11, 2023)

Interest rates are not the only parameters that can change -- other notable ones include changing liquidation ratios and even permitted collateral assets. The curious reader can peruse the governance proposals.

The GIF below shows how the Configuration smart contract is structured and how governance deploys new Comet instances with updated parameters.



CometConfiguration.sol and CometStorage.sol

CometConfiguration.sol defines the struct which parameterize the entire behavior of Comet. CometStorage simply stores those structs.

If you've read our article on How DeFi Interest Rates Work and DeFi Collateral and Liquidations, then hopefully the variable names in the struct are mostly self-explanatory. ConfiguratorStorage inherits these definitions and stores the structs in mappings.

These storage variables are not part of Comet. The Configurator contract deploys Comet instances using these stored configurations.

Below we show a snapshot of the code for CometConfiguration and ConfiguratorStorage.

This is a much more preferable way to deploying new Comet instances instead supplying an extremely large struct in the calldata.

When a new Comet instance is deployed with an update, we only need to update a particular storage variable, leaving the others unchanged. For example, if we change the liquidation threshold for the collateral wBTC, only that storage variable in the configurator will be affected.

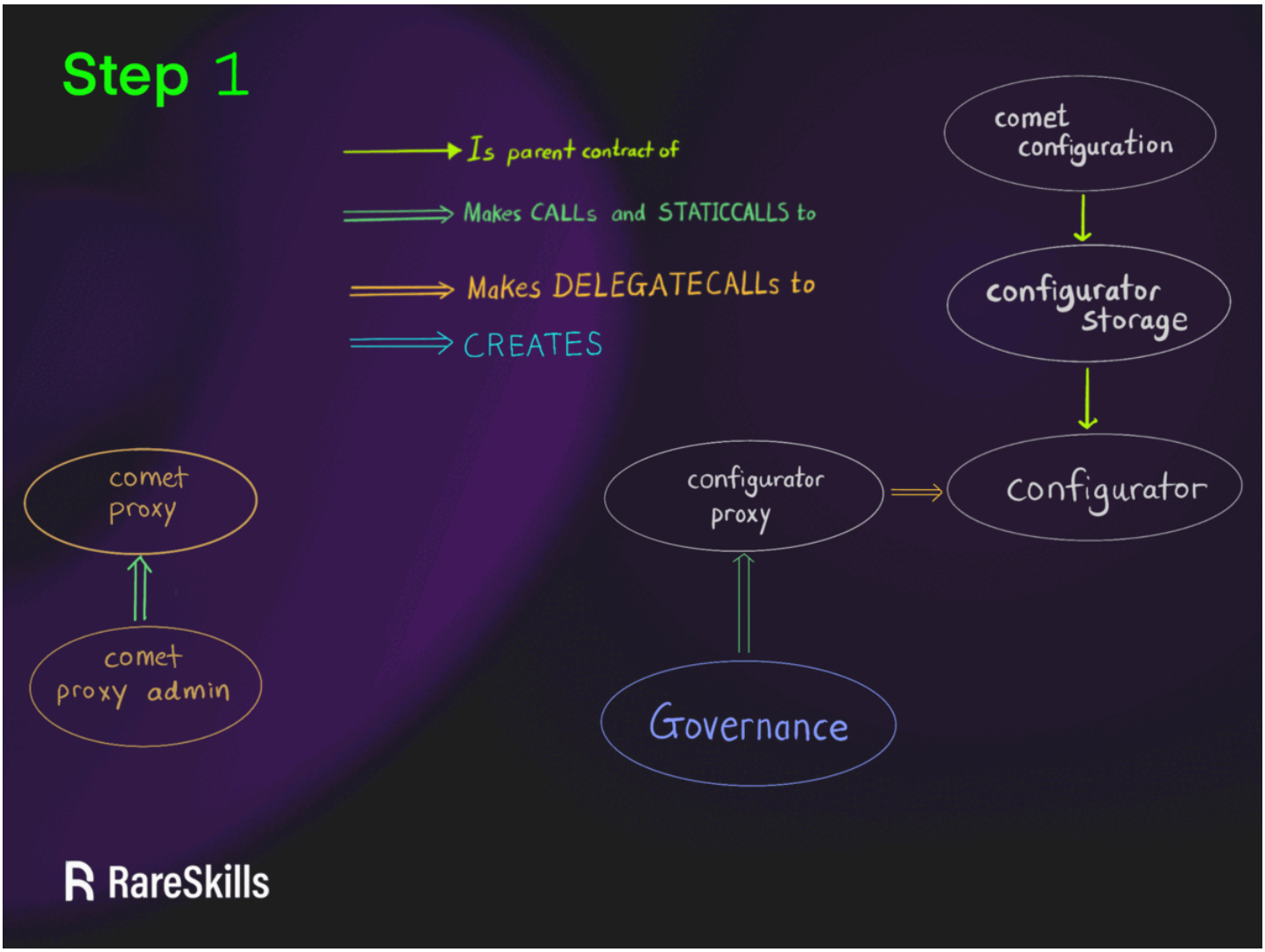
Configurator.sol

The Configurator.sol contract inherits ConfigurationStorage and provides governance-only setter functions. The Configurator.sol file is quite large, so we don't show the whole thing. However, just by looking at the events defined in it, we can get a good idea of what the contract does — update individual fields in the struct that parameterize the deployment of a new Comet instance.

Configurator.sol also holds the deploy() function to deploy new Comet instances.

CometFactory in the code above is defined in CometFactory.sol and is quite small, so we show the file in its entirety. The function "clone" is a bit misleading because it does not create proxy clones -- it creates a new instance.

We refer to the same animation from earlier to tie all of our previous discussion together.



Real Example of a Parameter Change

Let's use governance Proposal 162 as an example. We see it called some setter functions on the Configurator to update the parameters, then deployed a new Comet instance in the final step (step 8).

All Together

Below we show the relationship between all the files and how they interact with each other. The interface files are ignored.

Learn more with RareSkills

Please see our blockchain bootcamp to learn more.