How to Design a Lending Protocol on Ethereum

Comparing the Architecture of MakerDAO, Yield, Compound, Aave and Euler

Alberto Cuesta Cañada

Follow

--

Listen

Share

Borrowing is a cornerstone of Ethereum-based blockchain applications

. With billions in assets loaned out, understanding how borrowing works is crucial for developers, architects, or researchers.

Much like the evolution of programming paradigms, these DeFi applications have diverse architectural designs, reflecting changing priorities that range from security to efficiency and user experience.

This analysis looks at the architecture of applications like MakerDAO, Compound, Aave, Euler, and Yield. We'll highlight key innovations and design patterns, which are important lessons for the development of future lending applications.

If you're a developer, architect, or security researcher, this article is for you. By the end, you'll easily understand new borrowing applications on Ethereum, grasping their architecture swiftly and comprehensively. Dive in to see how these DeFi giants are built from the ground up.

Borrowing in DeFi

Most DeFi borrowing is overcollateralized. A user can borrow a specific asset if they provide collateral worth more than the loan. Unlike conventional loans, many of these loans don't have regular repayments or fixed end dates. In essence, you can borrow and never repay.

However, there is a catch.

The collateral's value must always exceed the loan value by a predetermined margin

.

If the collateral value falls below this, the loan isliquidated.

During liquidation, someone else repays part or all of your loan, and they receive part or all of your collateral in return.

All borrowing applications following this financial structure need the same building blocks, which then can be arranged in many ways:

- A treasury to store user collateral and borrowed assets
- An accounting system that tracks each user's collateral and debt
- Functions that determine the borrowers' interest rate
- A mechanism to verify if a loan is sufficiently collateralized, usually involving external price oracles
- A liquidation path for undercollateralized loans
- Risk management systems that record total borrowed amounts and other safety metrics, such as global and per-user borrowing limits, collateral minimums, and specific over-collateralization ratios
- An interface for users to add and remove collateral, borrow, and repay underlying

Borrowing and lending can be thought of as separate features. In DeFi, we find both features in most borrowing applications, but they are not always well integrated

.

In Compound, Aave and Euler they are

. The interest rates for borrowers and lenders are internally correlated; in fact, that is what makes those applications work with minimal intervention.

On the other hand, MakerDAO and Yield

are originators of the assets they lend to borrowers. They don’t require users to supply assets so that other users can borrow

.

This article will focus on on-chain borrowing and largely ignore lending. Borrowing is much more complicated because of the collateralization requirement, and understanding the borrowing pattern typically unlocks a better understanding of the entire protocol.

The Architectural Evolution of MakerDAO

MakerDAO, ancient in Ethereum terms, was launched in its current form in November 2019, and it holds $4.95B in collateral. Despite its modular architecture with distinct contracts for every function and unique terminology, it remains easy to understand and verify.

The treasury function in MakerDAO is managed by the

[Join

](https://github.com/makerdao/dss/blob/master/src/join.sol?ref=hackernoon.com)contracts.

There is a separate contract for each token approved as a collateral asset.

Contrarily, MakerDAO doesn’t possess any DAI, the borrowing asset.

Instead, it merely mints and burns DAI as required.

Accounting is handled within the vat.sol contract. The Joins update this contract when collateral enters or exits the system. If a user borrows, they interact with the vat.sol contract directly.

This action updates the user’s debt balance and allows them to mint DAI at the DAI Join.

To repay, users burn DAI in the DAI Join. This process then updates the Vat, enabling the user to settle their loan

.

Additionally, the vat.sol

contract serves as the risk management engine. It maintains global borrowing limits, sets per-user minimum thresholds, and oversees collateralization ratios.

When changes are made to a user’s debt or collateral balance, the vat.sol contract evaluates both rate and spot.

These refer to the interest rate based on the collateral used and the prevailing DAI-to-collateral price ratio. Interestingly, these values are fed into the vat.sol contract by other MakerDAO contracts, a method distinct from most other applications.

MakerDAO prioritized safety during its design phase — a time when factors like gas costs were secondary

, user experience was a minor concern, and competition was negligible.

Consequently, it might appear quirky, costly to use, and challenging to navigate.

Yet, the vast assets it manages and its record of operation without significant breaches underline its robust design and execution.

Highlights:

- Each asset has its own contract in the maximally spread treasury function
- The accounting function is centralized within a single contract that also documents and enforces risk parameters, including collateralization checks
- Unlike other apps, oracles update the contract, overseeing collateralization
- Price and interest rate oracles utilize distinct interfaces
- The interest rate originates externally

- To borrow, users must interact with multiple contracts

The Architectural Evolution of Yield Protocol

Yield v1 served as a proof of concept for fixed rates usingYieldSpace. This version built its collateralized debt engine on top of MakerDAO. However, Yield v1 was both expensive to use and challenging to augment with new features.

Recognizing the potential of YieldSpace, we quickly transitioned into developingYield v2. Still drawing inspiration from MakerDAO, but now completely independent, Yield v2 launched in October 2021; Yield v2 prioritized reduced gas costs and an enhanced user experience.

All accounting, risk management, and collateralization checks were consolidated into one contract: theCauldron. Mirroring MakerDAO's approach, we distributed the treasury functions across Join contracts, each dedicated to a specific asset.

We revamped our oracle integration, merging price and interest rate oracles into acommon interface. We reversed the oracle flow from MakerDAO such that Cauldron consults oracles as needed for collateralization checks. To my knowledge, this is the preferred flow everywhere except MakerDAO.

Another significant deviation from MakerDAO's approach was our introduction of theLadle. This contract serves as the sole intermediary between users and Yield. It wields extensive control over treasury and accounting, but in return, offers immense flexibility for feature development

.

In summary, borrowing in Yield v2 works as follows:

- Each asset has its own dedicated treasury contract, ensuring maximum distribution of the treasury function.
- A singular contract centralizes the accounting function. This contract also oversees risk management measures and performs collateralization checks.
- The collateralization function consults oracles to determine prices and interest rates.
- Both price and interest rate oracles share a unified interface.
- Interest rates are generated externally.
- Users can borrow by making a single request to just one contract.

The Architectural Evolution of Compound Finance

The first version of Compound was a Proof-of-Concept, demonstrating that a money market could be established on Ethereum. For this reason, simplicity was prioritized in its design. The MoneyMarket.sol contract encapsulates all functions, including lending.

- The treasury, accounting, and risk management tasks, such as collateralization checks, are consolidated into one contract.
- This contract retrieves prices from the oracles but determines interest rates based on asset utilization.
- Users interact solely with this contract, though they must make separate calls to supply collateral and borrow assets.

Compound v2

Compound v2 was launched in May 2019, igniting the yield farming era and inspiring countless forks. It too functioned as a money market, allowing users to both lend and borrow assets.

Based on its whitepaper and structure, it's evident that a major goal for Compound v2 was to use the ERC20 standard to represent lending positions. This ensured composability, allowing users to lend to Compound and then use those interest-bearing positions in other blockchain applications.

Interestingly, the whitepaper didn't highlight that Compound v2 incorporatedrewards into its smart contracts. Given this omission, the immense impact of this feature might not have been foreseen.

- Each asset has its own treasury contract, maximizing the distribution of the treasury function.
- The accounting function is also distributed, with each cToken noting user collateral and debt.

- A singular contract, the Comptroller, logs and enforces risk management parameters, including collateralization checks.
- The contract responsible for collateralization checks references the oracles for prices and the cToken for interest rates.
- The price and interest rate oracles operate with different interfaces.
- The interest rate derives internally from asset utilization.
- Users must interact with multiple contracts to borrow.

Compound v3

Released in 2022, Compound v3 adopts a more conservative risk management strategy, segregating the liquidity into a pool for each borrowable asset. The design also reveals concerns about user-friendliness and gas costs.

The system is more intuitive for both developers and users due to the reduction in the number of required calls. Additionally, the singular contract design reduces gas costs by minimizing calls between contracts. The segregated money markets are a defense against oracle-based attacks, which are now a major security concern.

Other relevant features mentioned in the release notes include:

- A completely revamped risk management and liquidation engine. This design enhances fund security while being more borrower-friendly.
- Set limits across the market for individual collateral assets to mitigate risks.
- The interest rate models for earning and borrowing are now separate, with governance having total control over economic policies.

Interestingly, Compound v3 mirrors the architecture of Compound v1 by having a single contract handle all functions for each borrowable asset. Other notable features include:

- Only lent assets can be borrowed; collateral assets cannot.
- Collateral does not yield returns in Compound v3.

The prohibition against borrowing collateral enhances the safety for those depositing the collateral. This decreases the likelihood that governance errors or intentional attacks jeopardize the collateral.

Eliminating returns on supplied collateral might be a result of Compound managing to amass much liquidity in v2. I have the intuition that in Compound v2 the borrowing limits were either below or not much higher than the assets lent to the application by the users.

Assuming they will manage similar levels of liquidity for v3, disallowing collateral to be lent out makes the application safe, one of the core goals of v3.

From an architectural standpoint:

- Every money market is an individual contract with its treasury, accounting, and risk management
- Each money market retains the borrowable asset along with all its approved collateral asset tokens, causing assets to be dispersed across the application
- Price feeds are the sole external input; interest rates for borrowing and lending are generated internally
- Traditional functions like supply/withdraw/borrow/repay have been smartly consolidated. Now, withdrawing a borrowable asset from a money market implies borrowing, while supplying it indicates repayment or lending based on the user’s debt
- A routing contract is integrated, permitting multiple operations in a single call

The Architectural Evolution of Aave

Aave v1 was launched in October 2019, succeeding ETHLend. Instead of ETHLend’s peer-to-peer approach, Aave v1 introduced a shared liquidity pool.

As in Yield v2, the router contract also held the business logic. The LendingPoolCore implemented accounting, risk management and treasury functions. Pooling the treasury in a single contract was a differentiating point from Compound v2.

The decision to leave the collateralization checks inits own contract, called from the router and not the accounting contract seems weak, but it was probably fit for purpose as Aave v2 was only released two years after the v1 release

- The LendingPoolCore contract handles treasury and accounting
- LendingPoolDataProvider manages collateralization checks and interacts with the oracle
- LendingPool serves as the user entry point and implements business logic
- Interest rates for borrowing and lending are determined internally, relying solely on price feeds

Aave v2

Aave v2 was released in December 2021. While it retained features similar to Aave v1, it introduced an improved and simpler architecture compared to both Aave v1 and Compound v2. With this release, Aave also introduced aTokens (akin to Compound's cTokens) and vTokens, which represent tokenized debt.

Certain features from Aave v1, which had limited use, were omitted for simplicity. Issues in Aave v1, like the complex representation of accrued interest, were addressed in Aave v2.

- The LendingPool contract consolidates global accounting and risk management functions, such as collateralization checks. It serves as the primary access point for users
- aTokens signify collateral and are akin to lending positions. Users' collateral is reflected through their aToken holdings, and the treasury function is distributed across all aTokens
- vTokens are used to represent debt positions. A user's debt is represented by the vTokens they hold

Aave v3

Aave v3 was released in January 2023 with multi-chain support and other features. These additions don't alter the core architecture. The update also boasts improved risk management and gas efficiency.

Despite its many advancements, for the purposes of this study, Aave v3 is not materially different from Aave v2. In fact, it might suggest that the architecture of Aave v2 remains robust in 2023.

The Architectural Evolution of Euler

Euler was launched in December 2022, aiming to offer money markets with permissionless features and minimal governance.

A hallmark of its design is thediamond-like pattern. A single contract holds all the application's storage. This storage can be accessed through distinct proxies, each managing a different conceptual element of the system.

Even though one contract stores all assets, accounting, and risk management data, there are still eTokens for collateral and lending, and dTokens for debt, similar to Aave v2. However, these token contracts are merely views of the central storage contract.

- The Storage contract manages accounting variables.
- The BaseLogic contract functions as the treasury.
- The RiskManager contract oversees risk management variables and functions, including collateralization checks.

An analysis of the code reveals that minimal gas cost was a priority, leading to the monolithic design eliminating the need for inter-contract calls. Security was assured through rigorous testing and auditing. Only the logic was distributed across various modules, serving as the implementation for the storage contract, which acted primarily as a proxy contract.

This unified design also supports easy upgrades. Modules can be swiftly replaced to amend or introduce features if no storage changes are required

.

Euler was hacked fifteen months after its release and six months after the upgrade introduced the exploited vulnerability.

I do not think the monolithic architecture played a part in the assets being drained; rather, it was insufficient oversight of code updates

.

Conclusion

Early Ethereum applications such as MakerDAO, Compound, and Aave showcased the potential of overcollateralized borrowing on Ethereum. Once these proofs of concept proved successful, the focus shifted to introducing a mix of new features to capture market share. Compound and Aave's later versions introduced yield farming, composability, and pooled liquidity, which thrived especially during bullish market conditions.

A significant development was Compound v2's introduction of tokenized lending positions, which enabled these positions to be recognized as standard assets by other applications. Aave v2 and Euler took it a step further by implementing tokenized debt positions, the broader utility of which remains a subject of debate.

High gas costs emerged as a major concern during the bull market, prompting user experience modifications as seen in Yield v2, Aave v2, and Euler. Router contracts and monolithic implementations helped reduce the costs users incurred for transactions. However, this came at the expense of more intricate, and consequently riskier, code.

Compound v3 appears to set a precedent, prioritizing safety over financial efficiency. It deviates from the traditional liquidity pool model to better safeguard against potential hacks. The rise of L2 networks, where gas costs are becoming increasingly negligible, will likely shape the design of future collateralized borrowing applications.

In this article, I've provided a comprehensive overview of the key collateralized borrowing applications on Ethereum. The approach I've employed to analyze each application can also be applied to grasp the intricacies of other collateralized borrowing applications swiftly.

When developing a blockchain borrowing application, always consider the storage of assets, the placement of accounting records, and the methods for risk and collateral evaluation. As you work through these considerations, draw upon the history of previous applications and the insights from this overview to inform your decisions.

Thank you for reading, and best of luck.

Thanks to

[Calnix

](https://twitter.com/cal_nix?ref=hackernoon.com)for reviewing and editing this article.