[

Aave v3.3_banner

1920×820 166 KB

](https://europe1.discourse-cdn.com/flex013/uploads/aave/original/2X/b/b5ec15f7e3b6f8ebab0b95497f65cf9a57b49bc4.jpeg)

Summary

Introducing Aave v3.3, a new version of Aave V3 (currently v3.2), upgrading the protocol debt management and liquidations, and making Aave fully compatible with the upcoming Umbrella.

Key changes include:

- Bad Debt Management

: bad debt logging & burn functionality to handle unresolvable debts, creating compatibility with the upcoming Umbrella's automated bad debt coverage.

- Liquidation Optimisations

: Refining the liquidations algorithm to reduce dust debt positions and make the procedure more economically appealing.

The current codebase (not all security procedures are finished on it) can be found onhttps://github.com/aave-dao/aave-v3-origin/pull/87.

Context

Aave v3.2 introduced Liquid eModes for greater capital efficiency, allowing assets to participate in multiple eModes at the same time. Aave v3.3 changes the target to a different module of the protocol, focusing on bad debt management and liquidation optimizations

.

The new bad debt management system introduces functionality to identify and precisely log bad debt, which occurs when liquidations leave accounts with zero collateral but outstanding debt, creating potential financial risks for the protocol. While the upcoming Umbrella upgrade (outlined in the [Umbrella Proposal]) will cover this logged bad debt, it remains the system's responsibility— Aave v3—to record where and how much bad debt exists accurately.

In addition, historically, the structure of liquidations often left minor residual debts that accumulated over time and became uneconomical to clear, creating inefficiencies and potential liabilities for the protocol. By improving the precision of the liquidation algorithm and reducing these dust debt positions, Aave v3.3 ensures that liquidations are more complete.

Aave v3.3. features

1. Deficit Management (Umbrella compatibility)

In Aave v3, certain liquidation scenarios can leave accounts with residual bad debt (outstanding debt without collateral), which continues to accrue interest and adds risk to the protocol's backing.

Historically, the Aave Safety Module (stkAAVE, stkABPT, stkGHO) has addressed this risk, but the system depends on subjective governance for slashing. With the upcoming Umbrella, slashing will be automated and governance-less, which requires introducing a bad debt management mechanism into the Aave v3 protocol.

The new bad debt management feature in v3.3 works the following way:

- During liquidation, a new validation step is introduced: the total collateral and total debt values of the account post-liquidation are checked, if an account ends up with zero collateral and non-zero debt, any remaining debt in the account is burned (bad debt cleanup

) and the bad debt created, called deficit

, is accounted to a new data field on the reserve data.

- As previously commented, the burning of bad debt and deficit accounting helps specifically in 2 ways:

- Allows Umbrella to understand how much deficit is to cover, objectively.
- “Discounts” bad debt for any debt growth accounting purposes, reducing liabilities to the protocol towards aToken holders.
- Allows Umbrella to understand how much deficit is to cover, objectively.
- “Discounts” bad debt for any debt growth accounting purposes, reducing liabilities to the protocol towards aToken holders.
- A new eliminateReserveDeficit()

function is introduced into the pool, permissioned to a UMBRELLA

role, that the core Umbrella smart contract in charge of slashing will own.

This will allow for 2 flows:

1. Burn aTokens slashed from the Umbrella staking.
2. In the special case of GHO on v3 Ethereum main, burn it directly, not having aToken.
3. Burn aTokens slashed from the Umbrella staking.
4. In the special case of GHO on v3 Ethereum main, burn it directly, not having aToken.
5. Bad debt/deficit is defined as strictly zero collateral in the reference currency of an Aave pool (USD). The goal of it is to be as objective as possible, without requiring any subjectivity/oracle from outside the system, defining more granularly what bad debt is.
6. For existing positions already with bad debt, this upgrade does not offer any solution but recommends the DAO to clean up these positions via a repayOnBehalf

. This simplifies substantially the upgrade, while achieving the same result.

2. Liquidations optimisation

The liquidation logic and library are pretty monolithic components of Aave by design: critical logic, audited countless times, battle-proven for years since v2. Development-wise, fully refactoring is an enormous effort and a highly strategic step (when/how).

However, for the aforementioned Deficit Management feature we needed to touch small parts of the logic, and together with it, decided to optimize different components following certain principles:

1. Avoid too invasive logic

. No matter if we think certain logic re-organization could be cleaner, similar to previous upgrades (e.g. stable rate), the goal of v3.3 is not changing major logic flows; definitely not on liquidations.

1. Target those aspects that deserve improvement, after historical analysis

. The Aave protocol has a very important track record on-chain and millions of users and interactions. That allows us to really understand empirically which optimizations are actually worth it and which ones are not.

1. Good harmony with Umbrella and its stakers

. Improving liquidations directly affects Umbrella: better liquidations → probabilistically less slash risk for stakers → less premium to pay in rewards by the Aave DAO in Umbrella.

With the previous in mind, Aave v3.3 includes the following improvements on liquidations.

2.1. Position-wide 50% Close Factor (CF)

The Aave protocol currently implements a so-called “Close Factor” which determines how much of a debt position can be repaid in a single liquidation. While in Aave v2, this parameter was a constant 50%, in Aave v3 the logic is slightly more sophisticated, and the close factor varies between a default 50% and a max close factor of 100%, currently applied when a user health factor deteriorates below a certain threshold (0.95HF).

The 50% close factor has always been applied on a per reserve basis, so if a user has e.g. the following position composition:

- 3k $ GHO Debt.

- 3k $ USDC Debt.
- 3k $ DAI Debt.
- 9k $ ETH Collateral

A liquidation would only be able to liquidate 1.5k $ GHO, USDC or DAI per liquidation, the 50% of one of the reserves.

While being by design, we believe this:

1. Is rather unintuitive

. It is an asset-specific resolution: the limitation applies to an amount of X asset, no matter the position composition. While global trigger: CF to apply is decided based on HF/overcollateralisation.

1. It is problematic in smaller positions

. If the overall position value falls below a certain threshold, it might no longer be economically sound to liquidate that position - by applying the close factor to each specific reserve, this problem scope is increased unnecessarily.

Therefore in Aave v3.3 the Close Factor is altered to apply for the whole position, which in the above example would allow to liquidate the whole 3k $ GHO/USDC or DAI in a single liquidation.

2.2. Extra cases to apply 100% Close Factor (CF)

For the Aave protocol, it is problematic to have dust debt positions, as there is no incentive to liquidate them depending on the network base cost, while on the other hand, they create an ever-increasing liability to the protocol.

From historical analysis, most of these dust debt positions are caused by the 50% close factor being applied to already small positions: if liquidators can only liquidate 50% of a position, this will keep decreasing the overall position value to a point where the gas cost no longer outweighs the liquidation bonus. As liquidators are usually very efficient, they trigger the action almost always above the 0.95HF threshold enforcing only 50%, and so the cascade effect.

Therefore to reduce the accumulation of minor debt positions, a new mechanism is introduced: liquidations up to a 100% close factor are now allowed whenever the total principal or the total debt of the user on the specific reserve being liquidated is below a MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD

Example

: Assuming a MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD

of $1’000 and a position composed as:

- 1200 $ collateral A
- 900 $ debt B
- A health factor at 0.96

In the previous system, a liquidation could have liquidated up to 50% of debt B

. With the new system, the debt position is below the MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD

. Therefore a liquidation could liquidate 100% of debt B

.

Acknowledged limitations

Liquidations are still highly influenced by gas prices, liquidation-bonus, and secondary market liquidity. MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD

has to be chosen in a “best effort” way to allow liquidations on “average” network conditions. A threshold of 2000$ for example would mean that, at a 1% bonus a liquidation cannot cost more than 20$ before no longer being liquidated by an economically reasonable liquidator.

2.3. Liquidation: Forced position cleanup

As elaborated before, small debt positions can be a burden already for the system, but they are especially problematic with the newly introduced bad debt cleanup:

- When leaving dust, the bad debt cleanup will not execute, as it only triggers when the collateral is exactly zero.

- As the bad debt cleanup slightly increases gas, even if minor, for liquidators there is an inherent incentive to leave dust.

To counter these problems a new mechanism is introduced to not allow any debt or collateral below MIN_LEFTOVER_BASE

to remain after a liquidation: if debt or collateral after the liquidation would be below MIN_LEFTOVER_BASE

, but none of the two is exactly zero, the transaction reverts.

To achieve that, MIN_LEFTOVER_BASE

is defined as MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD/2

. This way it is ensured that in the range of [0, MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD]

you can perform a full liquidation (via close Factor 100%). On the other hand, a 50% liquidation in the range of [MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD, Infinity]

will always leave at least MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD/2

.

Example

Assuming a MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD

of $1000, MIN_LEFTOVER_BASE

of $500, and a position composed as:

- $400 collateral A
- $1000 collateral B
- $900 debt A
- $400 debt B
- A health factor at 0.94

In the previous system, it would have been possible to liquidate any amount of debt for any respective amount of collateral. With the new system, you have to either:

- Liquidate 100% of debt B
- Liquidate 100% of collateral A
- Liquidate up to $400 of debt A

or liquidate 100% of debt A

- Liquidate up to $500 of collateral B

or liquidate 100% of collateral B

Acknowledged limitations

This feature is highly dependent on MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD

and therefore relies on choosing a reasonably high MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD

.

3. Miscellaneous

3.1. Bitmap access optimization

The current bitmasks on ReserveConfiguration

have been optimized for write operations. This is unintuitive, as the most common protocol interactions are read from the configuration during transactions.

By flipping the masks:

- uint256 internal constant LTV_MASK = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000; // prettier-ignore
- uint256 internal constant LTV_MASK = 0x000000000000000000000000000000000000000000000000000000000000FFFF; // prettier-ignore

The access can be simplified:

function getLtv(DataTypes.ReserveConfigurationMap memory self) internal pure returns (uint256) { - return self.data & ~LTV_MASK; + return self.data & LTV_MASK; }

Which slightly reduces gas & code size. The effect is getting more meaningful for accounts holding multiple collateral & borrow positions.

3.2. Additional getters

When analyzing ecosystem contracts we noticed that a lot of contracts have to pay excess gas due to the lack of fine-grained getters on the protocol: if an external integration e.g. wants to query the aToken balance of an address, it currently has to fetch Pool.getReserveData().aTokenAddress

which will read 9 storage slots.

This is suboptimal, as the consumer is only interested in a single slot - the one containing the aTokenAddress

. Therefore we added getReserveAToken()

and getReserveVariableDebtToken()

getters reducing gas cost by up to ~16k gas dependent on the use case.

3.3. Full deprecation getReserveDataExtended()

On v3.2 we flagged for deprecation the pool.getReserveDataExtended()

getter introduced with Aave v3.1, as it was not optimal and introduced legacy potential for any future upgrade.

v3.3 fully deprecates it, and its usage must be replaced on applications for pool.getReserveData()

, pool.getVirtualUnderlyingBalance()

& pool.getVirtualUnderlyingBalance()

.

FAQ

- So, Aave v3.3 is Umbrella?

No. Aave v3.3 includes certain components (mainly deficit accounting and removal) required for Umbrella to operate, but Umbrella is a separate system that will be connected to Aave v3.3.

- Will the release be disruptive for liquidators?

No. Liquidations on Aave will stay high-level as they were until now, with the only effect being extra restrictions on edge cases involving relatively small positions.

Even if we will provide full technical documentation for anybody running liquidators' infrastructure to check whether they should optimize it, generally, almost no change should be needed.

- Does this upgrade have implications in terms of potential risk configurations?

That will depend on risk providers' recommendations, but the liquidation bonus will most probably need to be slightly adjusted to adapt to feature 2.1.

- Will the new deficit affect the exchange rate of aToken to underlying (currently 1:1, due to balance growing)?

No. The deficit is a completely separated accounting mechanism compared with aToken, variable token, or any other. This is aligned with how Aave works now, where bad debt is assumed to be temporary and is covered by the Safety Module. And will continue to be similar to Umbrella.

- Will any change of Miscellaneous affect me if I’m an integrator?

Only the deprecation of getReserveDataExtended()

if you migrated to using that function on v3.1. The rest of the changes will be transparent to anybody consuming the Aave pool.

Security Procedures

As with all Aave upgrades, extensive security audits and testing will be completed before any activation, with some of them already finished.

Detailed findings and finalized security documentation will be published before that.

Next steps

1. Create an ARFC Snapshot for the upgrade of all Aave v3 instances to Aave v3.3, after some days of discussion in this forum.
2. In parallel, finish the preparations of the AIP payload for the upgrade, and security procedures of v3.3, and publish the results.
3. On-chain AIP creation, for the Aave governance to upgrade all pool instances across all networks.