# zkSync fee model

This document will assume that you already know how gas & fees work on Ethereum.

On Ethereum, all the computational, as well as storage costs, are represented via one unit: gas. Each operation costs a certain amount of gas, which is generally constant (though it may change during upgrades).

zkSync as well as other L2s have the issue which does not allow to adopt the same model as the one for Ethereum so easily: the main reason is the requirement for publishing of the pubdata on Ethereum. This means that prices for L2 transactions will depend on the volatile L1 gas prices and can not be simply hardcoded.

## High-level description

zkSync, being a zkRollup is required to prove every operation with zero knowledge proofs. That comes with a few nuances.

### `gas_per_pubdata_limit`

As already mentioned, the transactions on zkSync depend on volatile L1 gas costs to publish the pubdata for block, verify proofs, etc. For this reason, zkSync-specific EIP712 transactions contain the `gas_per_pubdata_limit` field in them, denoting the maximum price in *gas* that the operator can charge from users for a single byte of pubdata.

For Ethereum transactions (which do not contain this field), it will be enforced that the operator will not use a value larger value than a certain constant.

### Different opcode pricing

The operations tend to have different “complexity”/”pricing” in zero knowledge proof terms than in standard CPU terms. For instance, `keccak256` which was optimized for CPU performance, will cost more to prove.

That’s why you will find the prices for operations on zkSync a lot different from the ones on Ethereum.

#### Single-instance vs Multi-instance circuits

zkSync’s proving system consists of multiple types of circuits. For instance, `Keccak` circuit is used for calculating a single round of `Keccak` function, `MainVM` circuit is used for each cycle of the virtual machine and can do simple operations, like addition, substraction, etc. There are many more types of them, but listing all of them here is out of scope of this document.

Most of these circuits can be spawned infinite number of times. Meaning that in theory it is possible on zkSync to calculate `keccak256` or addition infinitely many times. But there are some circuits that are *single instance,* i.e. only a limited amount of them is allowed within a single block. If the user’s transaction has used too many of them, there is no way to prove it (since the block would not be able to be proven). Thus, for better compatibility with Ethereum, we must ensure that the users will never exceed the limits with our opcode pricing. The `MAX_TRANSACTION_GAS_LIMIT` as well as the opcode prices in gas are set in such a way that guarantees that the users will never exceed the limit. However, this also means that, for instance, the `BLOCK_GAS_LIMIT` will always be larger than the `MAX_TRANSACTION_GAS_LIMIT` because typically transactions do not consume single instance circuits in large quantities and most of the gas are spent on other, more abundant resources.

### Different intrinsic costs

Unlike Ethereum, where the intrinsic cost of transactions (`21000` gas) is used to cover the price of updating the balances of the users, the nonce and signature verification, on zkSync these prices are *not* included in the intrinsic costs for transactions, due to the native support of account abstraction, meaning that each account type may have their own transaction cost. Some may even use more zk-friendly signature schemes or other kinds of optimizations to allow cheaper transactions for their users.

That being said, zkSync transactions do come with intrinsic costs, but they are mostly used to cover costs related to the processing of the transaction by the bootloader which can not be easily measured in code in real-time. These are measured via testing and are hard coded.

## Block overhead & limited resources of the block

In order to process the block, the zkSync team has to pay for proving of the block, committing to it, etc. Processing a block involves some operational costs as well. All of these values we call “Block overhead”. It consists of two parts:

- The L2 requirements for proving the circuits (denoted in gas).
- The L1 requirements for the proof verification as well as general block processing (denoted in L1 gas).

We generally try to aggregate as many transactions as possible and each transaction pays for the block overhead proportionally to how close did the transaction bring the block to being *sealed,* i.e. closed and prepared for proof verification and submission on L1. A transaction gets closer to sealing a block by using the block’s *limited resources*.

While on Ethereum, the main reason for the existence block gas limit is to keep the system decentralized & load low, i.e. assuming the existence of the correct hardware, only time would be a requirement for a block to adhere to. In the case of zkSync blocks, there are some limited resources the block should manage:

- **Time.** The same as on Ethereum, the block should generally not take too much time to be closed in order to provide better UX. To represent the time needed we use a block gas limit, note that it is higher than the gas limit for a single transaction.
- **Slots for transactions.** The bootloader has a limited number of slots for transactions, i.e. it can not take more than a certain transactions per block.
- **The memory of the bootloader.** The bootloader needs to store the transaction’s ABI encoding in its memory & this fills it up. In practical terms, it serves as a penalty for having transactions with large calldata/signatures in case of custom accounts.
- **Single-instance circuits.** Since there is a limited number of times a certain operation can be performed, the more transactions use up these resources, the more it should pay for the block’s overhead costs.
- **Pubdata bytes.** In order to fully appreciate the gains from the storage diffs, i.e. the fact that changes in a single slot happening in the same block need to be published only once, we need to publish all the block’s public data only after the transaction has been processed. Right now, we to publish all the data with the storage diffs as well as L2→L1 messages, etc in a single transaction at the end of the block. Most nodes have limit of 128kb per transaction and so this is the limit that each zkSync block should respect.

Each transaction spends the block overhead proportionally to how close it consumes the resources above.

Note, that before the transaction is executed, the system can not know how many of the limited system resources the transaction will actually take, so we need to charge for the worst case and provide the refund at the end of the transaction.

## How EIP1559 works on zkSync & the limitations that come from the limited number of pubdata per block

Note, that in order to protect us from DDoS attacks we need to set a limited `MAX_TRANSACTION_GAS_LIMIT` per transaction (in order to prevent overusage of the single-instance circuits as well as to generally prevent large transactions from stalling the system).

Since the computation costs are relatively constant for us, we *could* use a “fair” `baseFee` equal to the real costs for us to compute the proof for the corresponding 1 erg. Note, that `gas_per_pubdata_limit` should be then set high enough to cover the fees for the L1 gas needed to send a single pubdata byte on Ethereum. Under large L1 gas, `gas_per_pubdata_limit` would also need be large. That means that `MAX_TRANSACTION_GAS_LIMIT/gas_per_pubdata_limit` could become too low to allow for the pubdata needed for deployments of new contracts to be sent.

To make the deployment transactions always executable, we must enforce that the users are always able to send at least `GUARANTEED_PUBDATA_PER_TX` bytes of pubdata in their transaction. Because of that, the needed `gas_per_pubdata_limit` for transactions should never grow beyond `MAX_TRANSACTION_GAS_LIMIT/GUARANTEED_PUBDATA_PER_TX`. Setting a hard bound on `gas_per_pubdata_limit` also means that with the growth of L1 gas prices, the L2 `baseFee` will have to grow as well (to ensure that `base_fee * gas_per_pubdata_limit = L1_gas_price * gas_per_pubdata)`).

This does not actually matter a lot for normal transactions, since most of the costs will still go on pubdata for them. However, it may matter for computationally intensive tasks, meaning that for them a big upfront payment will be required, with the refund at the end of the transaction for all the overspent gas.

## High-level: conclusion

The zkSync fee model is meant to be the basis of the long-term fee model, which provides both robustness and security. One of the most distinctive parts of it is the existing of the block overhead, which is proportional for the resources consumed by the transaction.

The other distinctive feature of the fee model used on zkSync is the abundance of refunds, i.e.:

- For unused limited system resources.
- For overpaid computation.

This is needed because of the relatively big upfront payments required in zkSync to provide DDoS security.

# Formalization

After determining price for each opcode in gas according to the model above, the following formulas are to be used for calculating `baseFee` for a block.

### System-wide constants

These constants are to be hardcoded and can only be changed via either system contracts’/bootloader or VM upgrade.

`GAS_PER_CIRCUIT` ($E_C$) — The number of gas required to prove a single circuit. The price of a single unit of a mutliinstance circuit $MC_i = \lceil \frac{E_C}{CC_i} \rceil$, where $CC_i$ is the capacity of the ith circuit. The number $E_C$ is an arbitrary constant, but it should be selected with the fact that all the prices for circuits are discrete, i.e. if we select this value to be too low, then circuits with vastly different capacities (the $CC_i$ in the formula) may have the same cost, because of the ceiling of the equation.

`L1_GAS_OVERHEAD` ($L1_O$)— The L1 gas overhead for a block (proof verification, etc).

`L1_GAS_PER_PUBDATA_BYTE` ($L1_{PUB}$) — The number of L1 gas needed for a single pubdata byte. It is slightly higher than 16 gas needed for publishing a non-zero byte of pubdata on-chain (currently the value of 17 is used).

`CONSTANT_GAS_OVERHEAD` ($E_O$)— The constant overhead denominated in gas. This overhead is equal to the $E_C$ multiplied by the number of circuits + some additional overhead needed to operate the bootloader.

`BLOCK_GAS_LIMIT` ($B$) — The maximum number of computation gas per block. This is the maximal number of gas that can be spent within a block. This constant is rather arbitrary and is needed to prevent transactions from taking too much time from the state keeper. It can not be larger than the hard limit of $2^{32}$ of gas for VM.

`MAX_TRANSACTION_GAS_LIMIT` ($T_M$) — The maximal transaction gas limit. For $i$-th single instance circuit, the price of each of its units is $SC_i = \lceil \frac{T_M}{CC_i} \rceil$ to ensure that no transaction can run out of these single instance circuits.

`MAX_TXS_PER_BLOCK` ( $TX_M$) — The maximum number of transactions per block. A constant in bootloader. Can contain almost any arbitrary value depending on the capacity of block that we want to have.

`BOOTLOADER_MEMORY` ($B_M$) — The size of the bootloader memory that is used for transaction encoding (i.e. excluding the constant space, preallocated for other purposes).

`MAX_PUBDATA_PER_TX` ($P_M$) — The maximum number of pubdata that can be sent within zkSync block.

`GUARANTEED_PUBDATA_PER_TX` ($P_G$) — The guaranteed number of pubdata that should be possible to pay for in one zkSync block. This is a number that should be enough for most reasonable cases, e.g. contract deployments.

## Derived constants

Some of the constants are derived from the system constants above:

`MAX_GAS_PER_PUBDATA` ( $EP_{Max}$) — the `gas_price_per_pubdata` that should always be enough to cover for publishing a pubdata byte:

$$EP_{Max} = \lfloor \frac{T_M}{P_G} \rfloor$$

## Externally-provided block parameters

`L1_GAS_PRICE` ($L1_P$) — The price for L1 gas in ETH.

`FAIR_GAS_PRICE` ($E_f$) — The “fair” gas price in ETH, that is, the price of proving one circuit (in Ether) divided by the number we chose as one circuit price in gas.

$$E_f = \frac{Price_C}{E_C}$$

where $Price_C$ is the price for proving a circuit in ETH. Even though this price will generally be volatile (due to the volatility of ETH price), the operator is discouraged to change it often, because it would volatile both volatile gas price and (most importantly) the required `gas_price_per_pubdata` for transactions.

Both of the values above are currently provided by the operator. Later on, some decentralized/deterministic way to provide these prices will be utilized.

## Determining base_fee

When the block opens, we can calculate the `FAIR_GAS_PER_PUBDATA_BYTE` ($EP_f$) — “fair” gas per pubdata byte:

$$EP_f = \lceil \frac{L1_p * L1_{PUB}}{E_f} \rceil$$

There are now two situations that can be observed:

1. $EP_f > EP_{Max}$

This means that the L1 gas price is so high that if we treated all the prices fairly, then the number of gas required to publish guaranteed pubdata is too high, i.e. allowing at least $P_G$ pubdata bytes per transaction would mean that we would to support $tx.gasLimit$ greater that the maximum gas per transaction $T_M$, allowing to run out of other finite resources.

If $EP_f > EP_{max}$, then the user needs to artificially increase the provided $E_f$ to bring the needed $tx.gasPerPubdataByte$ to $EP_{max}$

In this case we set the EIP1559 `baseFee` ($Base$):

$$Base = max(E_f, \lceil \frac{L1_P * L1_{PUB}}{EP_{max}} \rceil)$$

Only transactions that have at least this high gasPrice will be allowed into the block.

2. Otherwise, we keep $Base = E_f$.

Note, that both cases are covered with the formula in case (1), i.e.:

$$Base = max(E_f, \lceil \frac{L1_P * L1_{PUB}}{EP_{max}} \rceil)$$

This is the base fee that will be always returned from the API via `eth_gasGasPrice`.

## Inclusion of a transaction

Let's define:

$B_U$ — the number of computational gas already spent in a block (i.e. gas spend on either multiinstance or single instance circuits).

$P_U$ — the amount of public data already spent in a block.

For each single-instance circuit $i$, $C_{iu}$ — the % of this single instance circuit that has been already spent in the block.

$overhead\_gas(tx)$ — a function that returns the amount of the block overhead in gas the transaction should pay upfront. (It will be defined later)

A transaction will be included in a block if all of the following are true:

- $tx.maxFeePerErg \geq Base$ (the EIP1559 property)

- $tx.gasLimit - overhead\_gas(tx) \leq min(B - B_U, T_M)$ (transaction does not exceed the gasLimit for transactions or the one left for block)
- $tx.gasLimit / tx.gasPerPubdataByte \leq P_M - P_U$ (transaction cannot exceed the limit for pubdata).
- $Base * tx.gasPerPubdataByte \geq L1_{PUB} * L1_P$ (the transaction does compensate for all published public data).
- For each single-instance circuit $i$ the following should be true: $tx.gasLimit \leq T_M * (1 - C_{iu})$ (the transaction can not exceed the limit in any of the single-instance circuits).

This way the operator knows the transaction will definitely fit into the block.

These calculations are done offchain and are used by the operator only.

## Calculating overhead for a transaction

Let's define by $tx.actualGasLimit$ as the actual gasLimit that is to be used for processing of the transaction (including the intrinsic costs). In this case, we will use the following formulas for calculating the upfront payment for the overhead:

$$
\begin{gathered}
S_O = 1/TX_M \\
M_O(tx) = encLen(tx)/B_M \\
E_{AO}(tx) = tx.actualGasLimit/T_M \\
maxPubdataInTx(tx) = tx.gasLimit/tx.gasPerPubdata \\
P_O(tx) = maxPubdataInTx(tx)/P_M \\
O(tx) = max(S_O, M_O(tx), E_O(tx), P_O(tx))
\end{gathered}
$$

where:

$S_O$ — is the overhead for taking up 1 slot for a transaction

$M_O(tx)$ — is the overhead for taking up the memory of the bootloader

$encLen(tx)$ — the length of the ABI encoding of the transaction's struct.

$E_{AO}(tx)$ — is the overhead for potentially taking up the gas for singe instance circuits.

$P_O(tx)$ — the overhead for the possible amount of public data to be taken.

$O(tx)$ — is the total share of the overhead that the transaction should pay for.

Then we can calculate the overhead that the transaction should pay as the following one:

$$
\begin{gathered}
L1_O(tx) = \lceil \frac{L1_O}{L1_{PUB}} \rceil * O(tx) \\
E_O(tx) = E_O * O(tx)
\end{gathered}
$$

Where

$L1_O(tx)$ — the number of L1 gas overhead (in pubdata equivalent) the transaction should compensate for gas.

$E_O(tx)$— the number of L2 gas overhead the transaction should compensate for.

Then:

$$
overhead\_gas(tx) = E_O(tx) + tx.gasPerPubdata \cdot L1_O(tx)
$$

When a transaction is being estimated, the server returns the following gasLimit:

$$tx.gasLimit = tx.actualGasLimit + overhead\_gas(tx)$$

Note, that when the operator receives the transaction, it knows only $tx.gasLimit$. The operator could derive the $overhead\_gas(tx)$ and provide the bootloader with it. The bootloader will then derive $tx.actualGasLimit = tx.gasLimit - overhead\_gas(tx)$ and use the formulas above to derive the overhead that the user should've paid under the derived $tx.actualGasLimit$ to ensure that the operator does not overcharge the user.

### Note on formulas

For the ease of integer calculation, we will use the following formulas to derive the $overhead(tx)$ :

$B_O(tx) = E_O + tx.gasPerPubdataByte \cdot \lfloor \frac{L1_O}{L1_{PUB}} \rfloor$

$B_O$ denotes the overhead for block in gas that the transaction would have to pay if it consumed the resources for entire block.

Then, $overhead\_gas(tx)$ is the maximum of the following expressions:

1. $S_O = \lceil \frac{B_O}{TX_M} \rceil$
2. $M_O(tx) = \lceil \frac{B_O \cdot encodingLen(tx)}{B_M} \rceil$
3. $E_O(tx) = \lceil \frac{B_O \cdot tx.gasBodyLimit}{T_M} \rceil$
4. $P_O(tx) = \lceil \frac{B_O \cdot maxPubdataForTx}{T_M} \rceil$, where $maxPubdataForTx = \frac{tx.bodyGasLimit}{tx.gasPerPubdataByte}$

### Deriving `overhead_gas(tx)` from `tx.gasLimit`

The task that the operator needs to do is the following:

Given the tx.gasLimit, it should find the maximal `overhead_gas(tx)`, such that the bootloader will accept such transaction, that is, if we done by $O_{op}$ the overhead proposed by the operator, the following equation should hold:

$$O_{op} \leq overhead\_gas(tx)$$

for the $tx.bodyGasLimit$ we use the $tx.bodyGasLimit = tx.gasLimit - O_{op}$.

There are a few approaches that could be taken here:

- Binary search. However, we need to be able to use this formula for the L1 transactions too, which would mean that binary search is too costly.
- The analytical way. This is the way that we will use and it will allow us to find such an overhead in O(1), which is acceptable for L1 transactions.

Let's rewrite the formula above the following way:

$$O_{op} \leq max(S_O, M_O(tx), E_O(tx), P_O(tx))$$

So we need to find the maximal $O_{op}$, such that $O_{op} \leq max(S_O, M_O(tx), E_O(tx), P_O(tx)) \cdot B_O$. Note, that it means ensuring that at least one of the following is true:

1. $O_{op} \leq S_O$

2. $O_{op} \leq M_O(tx)$
3. $O_{op} \leq E_O(tx)$
4. $O_{op} \leq P_O(tx)$

So let's find the largest $O_{op}$ for each of these and select the maximum one.

▼ Solving for (1)

$$O_{op} = \lceil \frac{B_O}{TX_M} \rceil$$

▼ Solving for (2)

$$O_{op} = \lceil \frac{encLen(tx) \cdot B_O}{B_M} \rceil$$

▼ Solving for (3)

This one is somewhat harder than the previous ones. We need to find the largest $O_{op}$, such that:

$$O_{op} \leq \lceil \frac{tx.actualErgsLimit \cdot B_O}{T_M} \rceil$$
$$O_{op} \leq \lceil \frac{(tx.ergsLimit - O_{op}) \cdot B_O}{T_M} \rceil$$

Note, that all numbers here are integers, so we can use the following substitution:

$$O_{op} - 1 < \frac{(tx.ergsLimit - O_{op}) \cdot B_O}{T_M}$$
$$(O_{op} - 1)T_M < (tx.ergsLimit - O_{op}) \cdot B_O$$
$$O_{op}T_M + O_{op}B_O < tx.ergsLimit \cdot B_O + T_M$$
$$O_{op} < \frac{tx.ergsLimit \cdot B_O + T_M}{B_O + T_M}$$

Meaning, in other words:

$$O_{op} = \lfloor \frac{tx.ergsLimit \cdot B_O + T_M - 1}{B_O + T_M} \rfloor$$

▼ Solving for (4)

The formula for this one is the hardest

$$O_{op} \leq \lceil \frac{maxPubdataPerTx \cdot B_O}{T_M} \rceil$$
$$O_{op} \leq \lceil \frac{\lceil \frac{tx.txBodyErgsLimit}{tx.ergsPerPubdataByte} \rceil \cdot B_O}{T_M} \rceil$$

Note, that:

$$O_{op} \le \lceil \frac{\frac{tx.txBodyErgsLimit}{tx.ergsPerPubdataByte} \cdot B_O}{T_M} \rceil \le \lceil \frac{\lceil \frac{tx.txBodyErgsLimit}{tx.ergsPerPubdataByte} \rceil \cdot B_O}{T_M} \rceil$$

Instead of trying to solve the original inequality, we will solve the one in the middle. While it does marginally reduce the fee for the operator, the reasoning becomes a lot easier.

$$O_{op} \le \lceil \frac{\frac{tx.txBodyErgsLimit}{tx.ergsPerPubdataByte} \cdot B_O}{T_M} \rceil$$

Since $O_{op}$ is an integer, we can substitute with the following:

$$O_{op} - 1 < \frac{\frac{tx.txBodyErgsLimit}{tx.ergsPerPubdataByte} \cdot B_O}{T_M}$$
$$(O_{op} - 1)T_M < \frac{tx.txBodyErgsLimit}{tx.ergsPerPubdataByte} \cdot B_O$$
$$(O_{op} - 1)T_M \cdot tx.ergsPerPubdataByte < (tx.ergsLimit - O_{op}) \cdot B_O$$
$$O_{op}(T_M \cdot tx.ergsPerPubdataByte + B_O) < tx.ergsLimit \cdot B_O + T_M \cdot tx.ergsPerPubdataByte$$
$$O_{op} < \frac{tx.ergsLimit \cdot B_O + T_M \cdot tx.ergsPerPubdataByte}{B_O + T_M \cdot tx.ergsPerPubdataByte}$$

Or in other words:

$$O_{op} = \lfloor \frac{tx.ergsLimit \cdot B_O + T_M \cdot tx.ergsPerPubdataByte - 1}{B_O + T_M \cdot tx.ergsPerPubdataByte} \rfloor$$

Then, the operator can safely choose the largest one.

## Refunds

As you could see above, this fee model introduces quite some places where users may overpay for transactions:

- For the pubdata when L1 gas price is too low
- For the computation when L1 gas price is too high
- The overhead, since the transaction may not use the entire block resources they could.

To compensate users for this, we will provide refunds for users. For all of the refunds to be provable, the counter counts the number of gas that was spent on pubdata (or the number of pubdata bytes published). We will denote this number by $pubdataused$. For now, this value can be provided by the operator.

The fair price for a transaction is

$$FairFee = E_f * tx.computationalGas + EP_f * pubdataused.$$

We can derive $tx.computationalGas = gasspent - pubdataused * tx.gasPricePerPubdata$, where $gasspent$ is the number of gas spent for the transaction (can be trivially fetched in Solidity).

Also, the $FairFee$ will include the actual overhead for block that the users should pay for.

The fee that the user has actually spent is:

$$ActualFee = gasspent * gasPrice$$

So we can derive the overpay as

$$ActualFee - FairFee$$

In order to keep the invariant of $gasUsed * gasPrice = fee$ , we will formally refund $\frac{ActualFee - FairFee}{gasPrice}$ $\frac{ActualFee - FairFee}{Base}$ gas.

The operator is free to refund users with more funds (i.e. refunds for overhead).

# Problems to be solved in the upcoming releases

- Right now, the `gasPerPubdataByte` is practically a constant, because of the restrictions on the maximal number of gasPerPubdataByte.
- The additional area of research is to allow an unlimited amount of pubdata per block.
- The L1 gas price is provided by the operator, but it should be provided by some deterministic source.
- The protocol above does not answer the question of congestion fees. The reason d'etre for the EIP1559 was to correctly price the transaction under congestion, but what we have now is that the `baseFee` is mostly used to represent the price of L1 gas fee.