The zkEVM Architecture

Part V: Economics

Polygon zkEVM & Universitat Politècnica de Catalunya (UPC)

Marc Guzman-Albiol <marc.guzman.albiol@upc.edu>
Jose Luis Muñoz-Tapia <jose.luis.munoz@upc.edu>

Version: 0fd5d63ee0da211e457690d307873828c154069d

January 17, 2024

Outline

User Fees

Basic Ethereum Fee Schema i

The basic fee schema to which Ethereum users are used works as follows.

The gas is a unit that accounts the resources used when processing a transaction.

At the time of sending a transaction, the user can decide two parameters:

1. `gasLimit`:
    - It is the maximum amount of gas units that a user enables to be consumed by the transaction.
2. `gasPrice`:
    - It refers to the amount of Wei a user is willing to pay per unit of gas for the transaction execution.
    - In more detail, there is a market between users and network nodes such that if a user wants to prioritize his transaction, then he has to increase the `gasPrice`.

Basic Ethereum Fee Schema ii

- At the **start of the transaction processing**, the following amount of Wei is subtracted from the source account balance:

$$\texttt{gasLimit} \cdot \texttt{gasPrice}.$$

- Then,
  - If $\texttt{gasUsed} > \texttt{gasLimit}$, the transaction is reverted.
  - Otherwise, the amount of Wei associated with the unused gas is refunded.
- The refunded amount of Wei that is added back to the source account is calculated as:

$$\texttt{gasLimit} \cdot \texttt{gasPrice} - \texttt{gasUsed} \cdot \texttt{gasPrice}.$$

Generic User Fee Strategy of Layer 2 Solutions

- In general, Layers 2 follow the fee strategy of charging an L2 gasprice that is a percentage of the L1 `gasPrice`:

  $\texttt{L2GasPrice} = \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor}$.

- For example:

  $\texttt{L1GasPrice} = 20\ \text{Gwei}$
  $\texttt{L1GasPriceFactor} = 0.04\ (4\%\ \text{of L1}\ \texttt{gasPrice})$
  $\texttt{L2GasPrice} = 20\ \text{Gwei} \cdot 0.04 = 0.8\ \text{Gwei}$

- You can check the current fees at `https://l2fees.info`.

L2 User Fee Strategies are More Complex

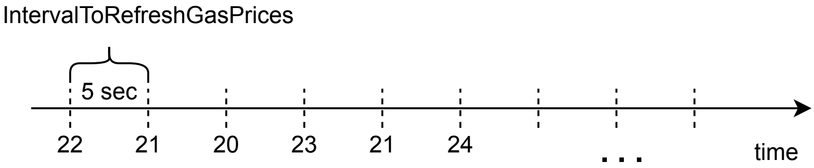
However, this is not as easy as it may seem and there are additional aspects to consider:

a) The `gasPrice` in L1 varies with time, so, how is this taken into account?

b) Different `gasPrice` values in L1 can be used to prioritize transactions, how are these priorities managed by the L2 solution?

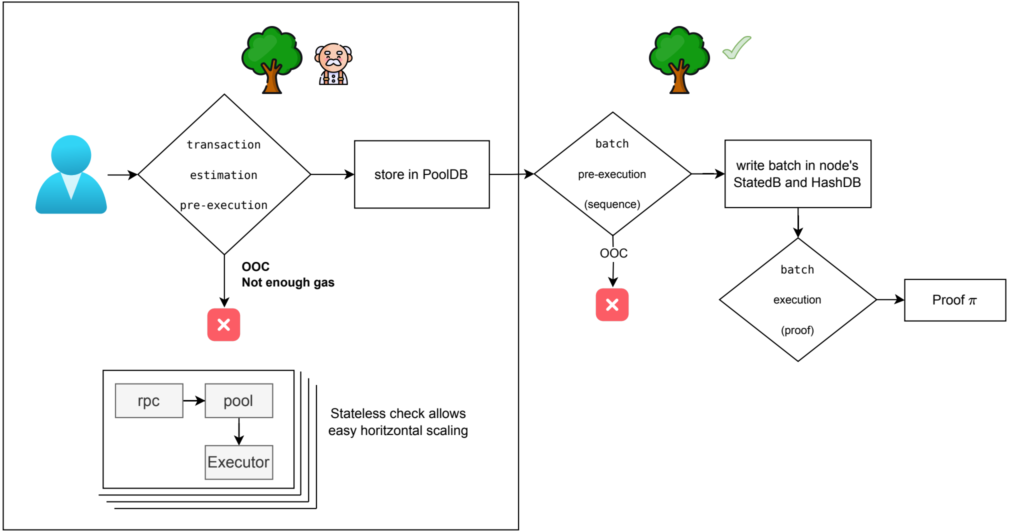
c) The `gas/gasPrice` L1 schema may not be aligned with the actual resources spent by the L2 solution.

Obtaining L1 GasPrices



In the example, we poll for the L1 gasPrice every 5 seconds and, as shown, gas prices vary with time.

RPC Transaction Pre-execution



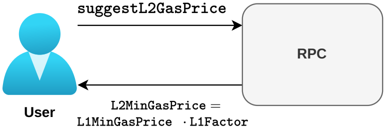
Sending a Transaction: User Experience

- We will operate in two steps: `gasPrice` suggestion and transaction sending.

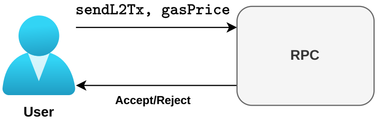
- In the first step **A**, the user will ask via RPC call for a suggested `gasPrice` computed as

$$\mathtt{L2GasPrice} = \mathtt{L1GasPrice} \cdot \mathtt{L1GasPriceFactor}$$

to sign its transaction with.



- In the second step **B**, the user sends the desired L2 transaction together with the `gasPrice`.

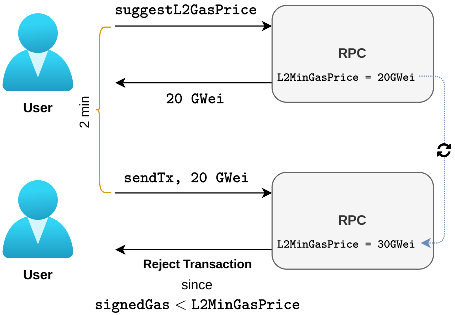


- If `gasPrice` provided by the user is less than the current `L2GasPrice`, the transaction is automatically **rejected**[a] and not included into the pool (error `ErrGasPrice`).

[a] Recall that the transaction can be rejected due to other checks explained before.

Sending a Transaction: Bad User Experience

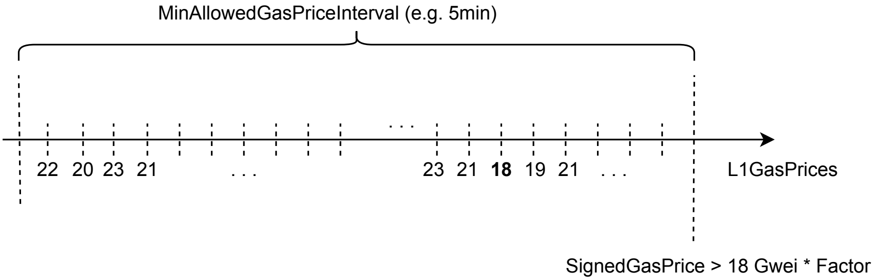
- However, this previous schema is not a good UX.
- Between steps A and B there is an unbounded interval of time.
- Henceforth, the `L2GasPrice` from step A can be different from the one present in step B, leading to the following unwanted situation:



- Observe that, since the `L2GasPrice` has been refreshed, the transaction sent by the prover will be rejected even tough it was signed with the exact suggested `gasPrice`.

Sending a Transaction: `MinAllowedPriceInterval`

The solution is to allow transactions from users that have signed any L2 `gasPrice` that is above the minimum L2 `gasPrice` recorded during a period of time (called `MinAllowedPriceInterval`).



Pool Config

We can configure the previous parameters in the Polygon zkEVM node:

```
[Pool]
...
DefaultMinGasPriceAllowed = 0
MinAllowedGasPriceInterval = "5m"
PollMinAllowedGasPriceInterval = "1s"
IntervalToRefreshGasPrices = "5s"
...
```

https://github.com/0xPolygonHermez/zkevm-node/blob/develop/docs/config-file/node-config-doc.md#75-pooldb

- `DefaultMinGasPriceAllowed`: It is the default min gas price to suggest.
- `MinAllowedGasPriceInterval`: It is the interval to look back of the suggested min gas price for a transaction.
- `PollMinAllowedGasPriceInterval`: It is the interval to poll L1 to find the suggested L2 min gas price.
- `IntervalToRefreshGasPrices`: It is the interval to refresh L2 gas prices.

Multi Gas Provider

When computing the L1 `gasPrice`, we can activate the `multigasprovider`:

```
[Etherman]
...MultiGasProvider = false
```

When enabled, it allows using multiples sources for computing the L1 `gasPrice`.

Gas Price Suggester

```
[L2GasPriceSuggester]
Type = "follower"
UpdatePeriod = "3s"
Factor = 0.12
DefaultGasPriceWei = 0
MaxGasPriceWei = 0
CleanHistoryPeriod = "1h"
CleanHistoryTimeRetention = "5m"
```

L1/L2 Costs Issues

- Gas in Ethereum accounts the resources used by the transaction.
- In particular, it takes into account:
  - **Data availability** (the transaction bytes).
  - **Processing resources**, like CPU, Memory and Storage.
- Ethereum users are used to priorize their transaction by increasing `gasPrice`.
- A big issue is that **there can be operations that consume low gas in L1 but that represent a major cost for L2**.
- The data availability costs are fixed once the transaction is known and they are directly proportional to L1 data availability costs.
- However, L2 execution is variable (because it depends on the state) and usually offers a smaller cost per gas.
- Henceforth, L2 transactions having high data availability costs and small execution costs are **highly problematic** in our pricing schema.

L1/L2 Costs Strategies

- Recall that the Ethereum fee is computed as `gasUsed` · `gasPrice`, giving us two ways of solving the misalignment problem:

  (A) **Arbitrum Approach. Increase `gasUsed`.**
  - This approach is based on changing the gas schema to increase the Gas costs for data availability.
  - This strategy is a relatively simple to implement and easy to understand but **it changes the Ethereum protocol**.
  - An L1 Ethereum transaction may execute different when compared to the same transaction executed in L2.

  (B) **Effective Gas Price Approach. Increase `gasPrice`.**
  - If we do not want to modify the Gas, we have to increase `gasPrice` in order to cover the costs.
  - Unlike the previous approach, this does not change the Ethereum specifications.
  - However, it is complex to achieve a fair `gasPrice`.
  - Moreover, we have to take into account that L2 users should be able to prioritize its transactions also increasing `gasPrice`, as they are used to.
  - This is actually our approach.

Effective Gas Price Overview i

- The user signs a relatively high gas price at the time of sending the L2 transaction.
- Later on, by pre-executing the sent transaction, the **sequencer** establishes a fair **gasPrice** according to the amount of resources used.
- To do so, the **sequencer** provides a single byte **effectivePercentageByte** $\in \{0, 1, \ldots, 255\}$ (1 Byte), which will be used to compute a ratio called **effectivePercentage**

$$\texttt{effectivePercentage} = \frac{1 + \texttt{effectivePercentageByte}}{256}.$$

- The **effectivePercentage** will be used in order to compute the factor of the signed transaction's **gasPrice** which should be charged to the user:

$$\texttt{txGasPrice} = \left\lfloor \texttt{signedGasPrice} \cdot \frac{1 + \texttt{effectivePercentageByte}}{256} \right\rfloor.$$

Effective Gas Price Overview ii

- For example, setting an `effectivePercentageByte` of $255 = $ `0xFF` would mean that the user would pay the totality of the `gasPrice` signed when sending the transaction:

$$\texttt{txGasPrice} = \texttt{signedGasPrice}.$$

- In contrast, setting `effectivePercentageByte` to 127 would reduce the `gasPrice` signed by the user to the half:

$$\texttt{txGasPrice} = \frac{\texttt{signedGasPrice}}{2}.$$

- Observe that, in this schema, users **must trust the sequencer**.
- As having `effectivePercentage` implies having `effectivePercentageByte`, and vice versa, we will abuse of notation and use them interchangeably as `effectivePercentage`.

About the `effectivePercentage` computation

- We could account the pricing resources by means of the number of **consumed counters** present in our proving system.
- Nevertheless, comprehending this can be challenging for users, and it is crucial to prioritize a positive user experience in this specific aspect.
- Moreover, stating the efficiency trough counters is not intuitive for users at the time of prioritizing their transactions.
- Henceforth, our actual goal is to compute `effectivePercentage` **only by using Gas** and prioritizing users transactions by means of using `gasPrice`.

Introduction of the `breakEvenGasPrice`

- Our goal as service providers is to **not accept transactions in which we loose money**.
- In order to achieve this, we will calculate the `breakEvenGasPrice`, considering a secure threshold to avoid losses in the event of unexpected issues.
- As explained before, we will split the computation in two to take into account differently costs associated with data availability and costs associated with used Gas.

`breakEvenGasPrice`: Costs Associated with Data Availability i

- Costs associated with Data Availability will be computed as

$$\texttt{dataCost} \cdot \texttt{L1GasPrice},$$

  where **`dataCost`** is the cost in Gas for data in L1.
- In the Ethereum ecosystem, the cost of data varies depending on whether it involves zero bytes or non-zero bytes

$$\texttt{NonZeroByteGasCost} = 16, \quad \texttt{ZeroByteGasCost} = 4$$

- In particular, **non-zero bytes** cost 16 **Gas** meanwhile **zero bytes** 4 **Gas**.

`breakEvenGasPrice`: Costs Associated with Data Availability ii

- Also recall that, when computing non-zero bytes cost we should take into account some constant data[1], always appearing in a transaction:
  - The **signature**, consisting on 65 bytes.
  - The **effectivePercentageBytesLength**, consisting on 1 bytes related to the RLP-encoded fields length.
- This results in a total of 66 constantly present bytes.
- Taking all in consideration, `dataCost` can be computed as:

$$(\texttt{txConstBytes} + \texttt{txNonZeroBytes}) \cdot \texttt{NonZeroByteGasCost} + \texttt{txZeroBytes} \cdot \texttt{ZerByteGasCost},$$

where `txZeroBytes` (resp. `txNonZeroBytes`) represents the count of zero bytes (resp. non-zero bytes) in the raw transaction sent by the user.

[1] This data can obtain zero bytes, but to optimize a little bit the processing we count them all of them as non-zero bytes.

`breakEvenGasPrice`: Computational Costs

- For the computational cost, we will simply use the following formula:

$$\texttt{gasUsed} \cdot \texttt{L2GasPrice},$$

  where recall that we can obtain `L2GasPrice` by multiplying `L1GasPrice` by chosen factor less than 1:

$$\texttt{L2GasPrice} = \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor}.$$

- In particular, we will choose a factor of 0.25

$$\texttt{L1GasPriceFactor} = 0.25.$$

- Observe that, unlike data costs, in order to compute computational costs we will need to **execute** the transaction.

effectiveGasPrice Formula

- Now, combining both **data** and **computational** costs, we will refer to it as `totalTxPrice`:

$$\texttt{totalTxPrice} = \texttt{dataCost} \cdot \texttt{L1GasPrice} + \texttt{gasUsed} \cdot \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor}.$$

- We can compute `breakEvenGasPrice` as the following ratio:

$$\texttt{breakEvenGasPrice} = \frac{\texttt{totalTxPrice}}{\texttt{gasUsed}}.$$

- This calculation helps to establish the gas price at which the total transaction cost is covered.
- Additionally, we incorporate a factor $\texttt{netProfit} \geq 1$ that allows us to achieve a slight profit margin:

$$\texttt{breakEvenGasPrice} = \frac{\texttt{totalTxPrice}}{\texttt{gasUsed}} \cdot \texttt{netProfit}.$$

- Observe that we still need to introduce here `gasPrice` prioritization.

Introducing Priority i

- Prioritization of transactions in Ethereum is determined by the signed `gasPrice`; higher values result in higher priority.
- To implement this, consider that users are only aware of two `gasPrice` values: the one signed with the transaction, called `gasPriceSigned`, and the one obtained from the RPC, that we will call `gasPriceSuggested`.

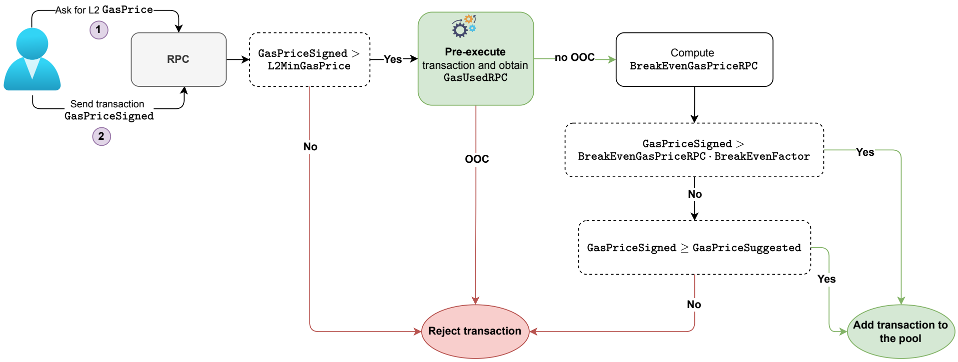
Introducing Priority ii

- In the case that $\texttt{gasPriceSigned} > \texttt{gasPriceSuggested}$, we establish a priority ratio as follows:

$$\texttt{priorityRatio} = \frac{\texttt{gasPriceSigned}}{\texttt{gasPriceSuggested}}.$$

- If $\texttt{gasPriceSigned} \leq \texttt{gasPriceSuggested}$, the user has chosen not to proritize its transaction (and maybe we can reject the transaction due to low gas price).
- In this case, we establish a priority ratio to be 1.
- The $\texttt{effectiveGasPrice}$ will be computed as:

$$\texttt{effectiveGasPrice} = \texttt{breakEvenGasPrice} \cdot \texttt{priorityRatio}.$$


Ask for L2 GasPrice
1
RPC
Send transaction GasPriceSigned
2
GasPriceSigned > L2MinGasPrice
Yes
No
Pre-execute transaction and obtain GasUsedRPC
no OOC
OOC
Compute BreakEvenGasPriceRPC
GasPriceSigned > BreakEvenGasPriceRPC · BreakEvenFactor
Yes
No
GasPriceSigned ≥ GasPriceSuggested
Yes
No
Reject transaction
Add transaction to the pool

`gasPrice` Flows: RPC ii

1. The user asks to the RPC for a suggested L2 `GasPrice`.
2. The users sends the transaction together with a selected `GasPriceSigned`.
3. The RPC pre-executes the transaction (**important**, using a wrong state root) to get all the execution-related parameters in order to compute the `breakEvenGasPrice`.
4. We have two cases here:
   - If the transaction pre-execution runs out of counters (**OCC** error), we immediately reject the transaction.
   - If not, the RPC computes the `breakEvenGasPrice` and we continue the flow.

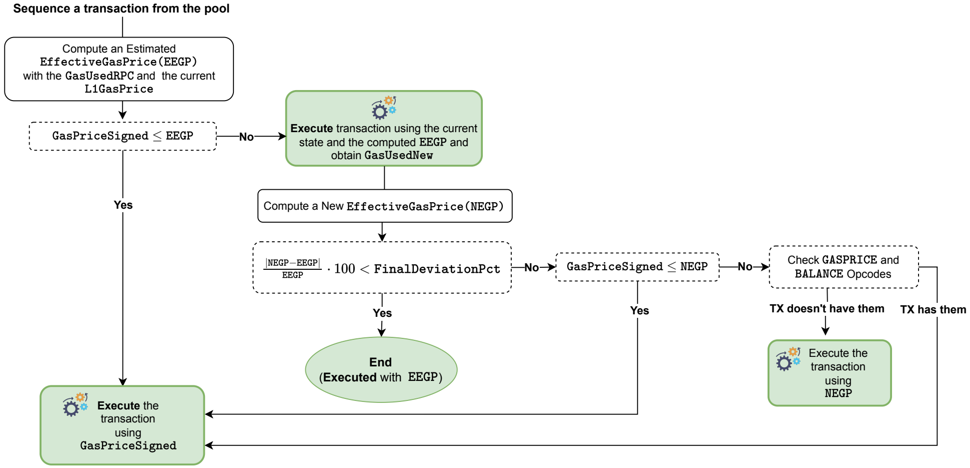
`gasPrice` Flows: RPC iii

5. Now, we have two options:
   - If $\texttt{gasPriceSigned} > \texttt{BreakEvenGasPrice} \cdot \texttt{BreakEvenFactor}$, we immediately accept the transaction, storing it in the pool.
   - The pool's transactions will be sorted by `effectiveGasPrice`, to take prioritization into consideration.
   - `BreakEvenFactor` (which is equal to 1.3) is introduced to provide a wider safeness threshold.
   - Otherwise $\texttt{gasPriceSigned} \leq \texttt{BreakEvenGasPrice} \cdot \texttt{BreakEvenFactor}$, we are in dangerous zone because we may be facing losings.
6. In the bad path, we allow two options:
   - If $\texttt{gasPriceSigned} \geq \texttt{gasPriceSuggested}$, we take the risk and we introduce the transaction into the pool.
   - Otherwise $\texttt{gasPriceSigned} < \texttt{gasPriceSuggested}$, we immediately reject the transaction because its **highly probable that we face losings**.

`gasPrice` Flows: RPC. Some Considerations

- It is important to remark that, **once a transaction is included into the pool, we should actually include it into a block**.
- Hence, if something goes bad in later steps and the processing consumes far more gas than expected, we will **loose money having no possibility to overcome that situation**.
- On the contrary, if the process goes well and the processing consumes less gas than expected, we can reward the user by modifying the previously introduced `effectivePercentage`.
- Also observe that all the transactions stored in the **Pool** should be ordered from larger to lower priorities (or, equivalently, `effectiveGasPrice`).

gasPrice Flows: Sequencer i



`gasPrice` Flows: Sequencer ii

1. The sequencer takes a transaction from the batch which is ready to be proved and recomputes the `EffectiveGasPrice` (which we will call `EGP`) with the execution-related data computed from the **raw state root** stored by the RPC and updated L2 gas prices.
2. At this point, we have two options:
   - If `gasPriceSigned` $<$ `EffectiveGasPrice`, there is a risk of loss.
   - In such cases, the user is charged the full `gasPriceSigned` and we end up the flow.
   - Conversely, if `gasPriceSigned` $\geq$ `EffectiveGasPrice`, there is room for further adjustment of the user's gas price.
3. In this case, we recompute a new `EffectiveGasPrice` (which we will call `NEGP`) with the execution-related data compute from the **correct state root**.

`gasPrice` Flows: Sequencer iii

4. We have two paths:
    - If the difference between EGP and NEGP is higher than some a parameter FinalDeviationPct (which is 10 in the actual configuration):

$$\frac{|\mathtt{NEGP} - \mathtt{EGP}|}{\mathtt{EGP}} \cdot 100 < \mathtt{FinalDeviationPct},$$

      we end up the flow just to avoid re-executions and save execution resources.
    - On the contrary, if the difference equals or exceeds the deviation parameter, there is a big difference between executions and we may better adjust gas price.
5. In the later case, two options arise:
    - If $\mathtt{gasPriceSigned} \leq \mathtt{NEGP}$ there is again a risk of loss.
    - In such cases, the user is charged the full gasPriceSigned and we end up the flow.
    - Otherwise, if $\mathtt{gasPriceSigned} > \mathtt{NEGP}$, means that we have margin to adjust the gas price.
    - However, we want to **save executions**, leading us to end up the process using a trick explained below.

6. We check if the transaction processing includes the two opcodes that use the gas price:
   - The `GASPRICE` opcode.
   - The `BALANCE` opcode from the source address.
7. If it is the case, to save one execution, we simply execute the transaction using the full `gasPriceSigned` to ensure we minimize potential losses and we end up the flow, as before.
8. If not, and with the intention of optimizing an execution while making a slight adjustment to the gasPrice, we proceed by executing the transaction using the `NEGP`.

Pool Effective Gas Price

```
...
    [Pool.EffectiveGasPrice]
    Enabled = false
    L1GasPriceFactor = 0.04
    ByteGasCost = 16
    ZeroByteGasCost = 4
    NetProfit = 1.2
    BreakEvenFactor = 1.3
    FinalDeviationPct = 10
    L2GasPriceSuggesterFactor = 0.3
```

https://github.com/0xPolygonHermez/zkevm-node/blob/develop/docs/config-file/node-config-doc.md

- `L1GasPriceFactor`: is the percentage of the L1 gas price that will be used as the L2 min gas price
- `ByteGasCost`: cost per byte that is not 0.
- `ZeroByteGasCost`: cost per byte that is 0.
- `NetProfit`: is the profit margin to apply to the calculated breakEvenGasPrice.
- `BreakEvenFactor`: is the factor to apply to the calculated breakevenGasPrice when comparing it with the gasPriceSigned of a tx.
- `FinalDeviationPct`: is the max allowed deviation percentage BreakEvenGasPrice on re-calculation
- `L2GasPriceSuggesterFactor`: is the factor to apply to L1 gas price to get the suggested L2 gas price used in the calculations when the effective gas price is disabled (testing/metrics purposes)