

Monad Initial Specification Proposal

Category Labs

Version 2.0.1

November 19, 2025

1 Introductory Concepts

1.1 Fields in Transactions and Blocks

A transaction t has the same fields as an Ethereum transaction. The relevant ones that the transaction specifies are:

- Gas limit, gas_limit_t .
Every transaction is only allowed to set $\text{gas_limit}_t \leq \text{max_tx_gas_limit}$, i.e., less than the full `block_gas_limit`.
- Type 2 (EIP-1559) transactions:
 - Priority fee per gas (“maxPriorityFeePerGas” in Yellow paper), $\text{priority_fee_per_gas}_t$.
 - Max fee per gas (“maxFeePerGas” in Yellow paper), max_fee_per_gas_t .
- Type 0 and type 1 transactions (legacy transactions):
 - Gas Price (“gasPrice” in Yellow paper), gas_price_t

The block header for block k contains three base fee-related fields:

- Base fee, base_fee_k
- Trend accumulator, trend_k , used inside the formula for the base fee update (details in Section 2.6).
- Moment accumulator, moment_k , used inside the formula for the base fee update (details in Section 2.6).

1.2 Asynchronous Execution

In Monad, block execution is allowed to be delayed with respect to consensus. Concretely, consensus block n includes the state root obtained by executing block $n - k$ where k is a protocol parameter defining the allowed execution delay. Consequently, validity checks performed by consensus for block n must only depend on state^{n-k} and the static data of block $n - k + 1$ through block n .

2 Transaction Fee Mechanism (TFM)

The bid for gas by transaction t in block k is given by:

$$\begin{aligned} \text{EIP-1559 tx type : gas_bid}_t &= \min(\text{priority_fee_per_gas}_t + \text{base_fee}_k, \text{max_fee_per_gas}_t) \\ \text{Legacy tx type : gas_bid}_t &= \text{gas_price}_t \end{aligned}$$

2.1 Inclusion and Block Construction

Validity checks for block k :

- Block gas limit check: $\sum_{t \in \text{block}} \text{gas_limit}_t \leq \text{block_gas_limit}$
- Check for every transaction t :
 - Base fee check: $\text{gas_bid}_t \geq \text{base_fee}_k$
 - Per-transaction gas limit check: $\text{gas_limit}_t \leq \text{max_tx_gas_limit}$
 - Other standard checks as in Ethereum, such as nonce matches the account nonce and gas limit covers the intrinsic gas.

2.1.1 Reserve Balance

Reserve Balance logic is performed in both the consensus and execution clients to ensure that despite asynchronous execution, sender accounts have enough balance to pay for gas fees at the time of execution. During execution, transactions revert due to spending of account balance outside of gas payments when the balance is below a specified reserve balance level. An exception is made for undelegated accounts that have no pending transactions within the past k blocks. When performing block validity checks for block n , consensus queries account balances of transaction senders from execution state as of block $n - k$ and checks that the worst-case balance for each sender after executing blocks $n - k + 1$ through block n will be non-negative when factoring in execution spending restrictions. See Section 6 for a full description of the consensus and execution logic.

2.2 Ordering

Transactions are ordered in the client according to their bid for gas, gas_bid_t . This ordering preference is not a validity check: it need not be checked on others' blocks. It only describes the default client implementation and is not meaningfully enforced by the protocol.

2.3 Payment Rule for User

The payment for transaction t is given by the gas fees:

$$\text{Payment}_t = \text{gas_bid}_t \cdot \text{gas_limit}_t$$

The user pays the gas fees on the gas limit, and there are no gas refunds (due to deallocations in `SSTORE` opcode for example). The user account is required to have enough balance to cover the upfront payment of $\text{gas_bid}_t \cdot \text{gas_limit}_t$ (more details in Section 6).

2.4 Burn Rule

The burn for each transaction t in block k is the gas burn:

$$\text{Burn}_t = \text{base_fee}_k \cdot \text{gas_limit}_t$$

2.5 Reward Rule for the Validator

The reward for the validator for block k is given by the sum of the priority gas fees and a constant block reward (staking and consensus-related rewards/penalties are not included here):

$$\text{Reward}_k = \sum_{t \in \text{block}} (\text{gas_bid}_t - \text{base_fee}_k) \cdot \text{gas_limit}_t$$

2.6 Base Fee Update Rule

Similar to EIP-1559, Monad has a mandatory base fee enforced by consensus that is computed algorithmically across blocks based on recent utilization relative to a target. As with block validity checks, utilization is measured with respect to gas_limit_{tx} . Additionally, a protocol parameter `min_base_fee` is introduced as a lower bound to the base fee paid. The update rule is defined as follows:

$$\begin{aligned} \text{block_gas}_k &= \sum_{tx \in \text{block}_k} \text{gas_limit}_{tx} \\ \text{base_fee}_{k+1} &= \max \left\{ \text{min_base_fee}, \text{base_fee}_k \cdot \exp \left(\eta_k \cdot \frac{\text{block_gas}_k - \text{target}}{\text{block_gas_limit} - \text{target}} \right) \right\} \\ \eta_k &= \frac{\text{max_step_size} \cdot \text{epsilon}}{\text{epsilon} + \sqrt{\text{moment}_k - \text{trend}_k^2}} \\ \text{trend}_{k+1} &= \text{beta} \cdot \text{trend}_k + (1 - \text{beta}) \cdot (\text{target} - \text{block_gas}_k) \\ \text{moment}_{k+1} &= \text{beta} \cdot \text{moment}_k + (1 - \text{beta}) \cdot (\text{target} - \text{block_gas}_k)^2 \end{aligned}$$

The initial values for the genesis block are: $\text{base_fee}_0 = 0$; $\text{moment}_0 = 0$; $\text{trend}_0 = 0$. The reference implementation is provided in Section 7; these are the integer math calculations that must be followed, including for example the flooring square root. For more discussion around the update rule and parameters, see Section 5.

3 List of Parameters with Suggestions

- Transaction-level:
 - `max_tx_gas_limit` = 30M (controls for large transactions)¹
- Higher protocol-level:
 - `block_gas_limit` = 200M (500M gas / sec for 0.4 second block times)
 - `block_size_limit` = 2MB (applies to total size of RLP encoded transactions; excludes header)
 - `min_base_fee` = 100 MON-gwei (i.e., $100 \cdot 10^{-9}$ MON)
 - `default_reserve_balance` = 10 MON (10 million gas in flight at 10 times the min base fee, yet small amount)
 - `k` = 3 (building / validating consensus block n requires knowing state obtained by applying block $n - k$)
- Base fee update rule:
 - Learning rate `max_step_size` = 1/28
 - Block gas `target` = 80% of limit = 160M
 - Smoothing for accumulators `beta` = 0.96
 - Scaling factor `epsilon` = $1 \cdot \text{target}$ = 160M

¹`max_tx_gas_limit` should be updated in conjunction with `target`.

4 Opcode and Precompile Gas Costs

The costs of a few opcodes differ from the Yellow paper.

4.1 Cold Access Cost

To account for the relatively higher cost of state reads from disk in the Monad execution client, the following changes are proposed to "cold" account and storage access costs:

Access Type	Ethereum	Monad
Account	100 if warm else 2600	100 if warm else 10100
Storage	100 if warm else 2100	100 if warm else 8100

Impacted opcodes:

- Account access: `BALANCE`, `EXTCODESIZE`, `EXTCODECOPY`, `EXTCODEHASH`, `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`, `SELFDESTRUCT`
- Storage access: `SLOAD`, `SSTORE`

For these opcodes, the dynamic access cost is unchanged for warm access and is as specified above for cold access.

4.2 Precompiles

Precompiles are recalibrated to accurately reflect their relative costs in execution, and the following precompiles' gas cost is changed compared to Ethereum:

Precompile	Address	Ethereum	Monad	Multiplier
<code>ecRecover</code>	0x01	3000	6000	2
<code>ecAdd</code>	0x06	150*	300*	2
<code>ecMul</code>	0x07	6000*	30000*	5
<code>ecPairing</code>	0x08	45000*	225000*	5
<code>blake2f</code>	0x09	rounds×1	rounds×2	2
<code>point eval</code>	0x0a	50000	200000	4

*: Per input/operation as defined in the respective precompile specification.

5 Discussion

5.1 Reasoning for Parameters

5.1.1 Block Gas Target and Goal of Update Rule

The intended behavior of the base fee mechanism is that the base fee only ramps up so long as there is long-term (i.e., not just for a few blocks) average gas usage close to the entire gas limit of the chain. Otherwise, and in the usual case that there is no sustained (long-term) demand above the block gas target, the base fee is intended to remain around `min_base_fee`. For this reason, the block gas target is set to 80% (significantly higher than 50% in Ethereum’s EIP-1559) to allow for higher utilization, yet leave room for collecting demand signals.

5.1.2 Minimum Base Fee

There are three ways to derive the minimum base fee:

- Cost to the users.
- The revenue/yield for the network, more specifically holders
- The marginal cost of processing an average unit of gas.

We use the first criterion. In order for a simple transfer (21,000 gas) to cost about 0.002 MON, set the `min_base_fee` to 100 MON-gwei (i.e., $100 \cdot 10^{-9}$ MON).

5.1.3 Base Fee Update Rule Parameters

First, an observation: by design, we want the fee to increase slower upon seeing full blocks than it decreases. We will design the learning rates for the former. Specifically, when seeing a series of empty blocks (depending on the smoothing parameter, longer or shorter), the fee will adjust at rates quadruple the ones we mention below; this is due to how we decided to set our `target`, and is an intended property to protect from overestimation of congestion relative to the block capacity.

We design the top end such that the highest rate corresponds to Ethereum’s rates but with 1 second block times, i.e., 3.6%. Then, we fix the lowest achievable rate to correspond to a less aggressive than Ethereum’s (half) at 1 second block times; or equivalently, Ethereum’s rate at 0.25 second block times.

These choices fully specify the parameters, and also, in hindsight, provide satisfying “normal” step size and step size lower bound if blocks are above the target: with a long-term variance of 20% of the block gas limit, this is a rate of 2.88%, corresponding to Ethereum’s rate with 0.64 second block times.

Finally, for the smoothing parameter, we choose a smoothing constant of $\beta = 0.96$ for a half-life of about 17 blocks, with the goal to definitely fade spikes of less than ~ 10 seconds, i.e., fade at least about 20 blocks.

5.2 Gas Limit vs Gas Used

Due to delayed execution, and to prevent denial of service speculative attacks (that would show up as severe underutilization of the chain’s capacity), the fees have to be charged on the transaction’s gas limit, or at least some function of it. We choose to charge **both** the base fee **and** the priority fee on the gas limit:

- For the base fee, charging on the gas limit penalizes more the variance in actual execution gas, making variance penalties on risk-adjusted arbitrage expected value (EV) heavier for speculative / optimistic arbitrageurs, thus disincentivizing these harmful use cases (over the execution of normal, less-variance user transactions). Such high-variance execution gas transactions are those that we define as more correlated with spam transactions that we would like to disincentivize.
- For the priority fee, charging on the gas limit allows validators to still accurately determine the fees paid to them under delayed execution. This is because it makes the knapsack problem of optimally choosing the transactions to include in a block as easy as it can be, and it helps in inducing a clear bidding behavior from users (upfront cost as a static unconditional payment). Payment based on gas

used would have been a conditional payment under delayed execution. Note that a conditional payment can already be made through coinbase transfers. Either kind of conditional payment (coinbase or based on gas used) would require synchronous execution (or accurate enough estimation) for the knapsack optimal transaction choice problem (of a validator) to be tractable.

5.3 Reserve Balance UX Implications

Reserve balance (Section 2.1.1) is introduced to address the unknown execution results of transactions pending in the last k blocks due to delayed execution. The design was chosen to minimize disruption to standard usage patterns while enabling advanced users to have multiple pending transactions. In terms of practical implications, reserve balance can cause a given transaction to:

- Be invalid for inclusion in a consensus block depending on the set of pending transactions.
- Revert in execution if an account balance decreases beneath the reserve balance outside of gas payments.

For an account that is undelegated (and with no pending authorizations), the first pending transaction is unaffected by reserve balance restrictions in either consensus or execution and is simply subject to standard balance solvency checks. Subsequent transactions sent from that account within k blocks could be impacted, especially if the account balance is close to or below the reserve level. For delegated accounts, all transactions are potentially affected.

6 Reserve Balance Specifications

There is a default reserve balance for all accounts denoted by `default_reserve_balance`. In a future version, the protocol could allow users, through a stateful precompile, to customize their reserve balance.

Denote the reserve balance for account a as `user_reserve_balance(a)`. Let the consensus-execution delay gap be denoted by k blocks. More specifically, building or validating a consensus block number n requires knowing about state obtained by applying block $n - k$ ($state^{n-k}$) from the execution client ($k > 1$ for asynchronous execution). Denote account a 's balance at a state st as `balance(a, st)`. Denote account a 's EIP-7702 delegation status at the state st as `is_delegated(a, st)`: it is true if the delegation is towards a non-null address, false otherwise. Let the block number in which tx is sequenced (or being considered for sequencing) be denoted by `blocknumber(tx)`.

We say a transaction is before (denoted by \prec ; \preceq denotes before or equal to) another transaction based on their ordering in the blockchain. Similarly, we say a transaction t is after a state s if t is after the transaction that produced s .

Let $\text{gas_fees}(t) = \text{gas_limit}_t \cdot \text{gas_bid}_t$.

Algorithm 1: Transaction Inclusion Validity. Used by consensus client and execution client with appropriate values of parameter x . For consensus client, $x = n - k$

Precondition: $n > x \geq n - k$

Given:

- t : Transaction from account $t.\text{sender}$ for block n
- state^x : State obtained by applying block x ($x = n - k$ for consensus client)
- \mathcal{I} : All intermediate transactions from account $t.\text{sender}$ from block $x + 1$ (inclusive) up to t (inclusive).
- \mathcal{P} : All transactions from block $x - k + 2$ (inclusive) up to t (inclusive).

Define:

$$\text{ValidInclusion}(t, \text{state}^x, \mathcal{I}, \mathcal{P}) := \tag{1}$$

$$\quad \text{let } \text{balance} = \text{balance}(t.\text{sender}, \text{state}^x) \tag{2}$$

$$\quad \text{let } \text{delegated} = \text{is_delegated}(t.\text{sender}, \text{state}^x) \tag{3}$$

$$\quad \text{let } \text{first_tx} = \mathcal{I}[0] \tag{4}$$

$$\quad \text{let } \text{starting_block} = \min(x + 1, \text{blocknumber}(\text{first_tx}) - k + 1) \tag{5}$$

Case Analysis:

$$\text{if } \text{first_tx} = t \wedge \text{IsEmptying}(\text{first_tx}, \mathcal{P}, \text{starting_block}, \text{delegated}) \text{ then} \tag{6}$$

(No intermediate transaction, don't do reserve balance checks if t qualifies)

$$\quad \text{balance} \geq \text{gas_fees}(\text{first_tx}) \tag{7}$$

$$\text{else} \tag{8}$$

$$\quad \text{if } \text{IsEmptying}(\text{first_tx}, \mathcal{P}, \text{starting_block}, \text{delegated}) \tag{9}$$

$$\quad \text{then} \tag{At most one emptying transaction: } \mathcal{I}[0]$$

$$\quad \quad \text{let } \text{adjusted_balance} = \text{balance} - (\text{first_tx}.\text{value} + \text{gas_fees}(\text{first_tx}))$$

(adjusted_balance can be negative here, implementation works with EVM's uint)

$$\quad \quad \text{let } \text{reserve} = \min(\text{user_reserve_balance}(t.\text{sender}), \text{adjusted_balance}) \tag{10}$$

$$\quad \quad \text{in } \text{reserve} \geq \sum_{\text{tx} \in \mathcal{I}[1:]} \text{gas_fees}(\text{tx}) \tag{Sum over all but first}$$

$$\quad \text{else} \tag{11}$$

$$\quad \quad \text{let } \text{reserve} = \min(\text{user_reserve_balance}(t.\text{sender}), \text{balance}) \tag{12}$$

$$\quad \quad \text{in } \text{reserve} \geq \sum_{\text{tx} \in \mathcal{I}} \text{gas_fees}(\text{tx}) \tag{Use reserve balance for all transactions}$$

Algorithm 2: Transaction Execution

Given:

- t : Transaction from account a in block n
- $state$: Current state right before executing t
- $prior_transactions$: All transactions from block $n - k + 1$ (inclusive) up to t (inclusive).

Define:

$$\text{ExecutionResult}(t, state, prior_transactions) := \quad (13)$$

$$\quad \text{let } sender = t.sender \quad (14)$$

$$\quad \text{let } solvency_condition = \text{balance}(sender, state) \geq \text{gas_fees}(t) \quad (\text{Soundness assertion})$$

$$\quad \text{let } (new_state, original_balances) = \text{Execute}(state, t) \quad (15)$$

$$\quad \text{let } delegated = \text{is_delegated}(sender, state) \quad (16)$$

$$\quad \text{let } is_emptying = \text{IsEmptying}(t, prior_transactions, n - k + 1, delegated) \quad (17)$$

$$\quad \text{let } reserve_dipped = \text{DippedIntoReserve}(original_balances, \quad (18)$$

$$\quad \quad \quad new_state, t, is_emptying) \quad (19)$$

$$\quad \text{let } reverted_state = \text{RevertExecution}(state, t)$$

$$\quad (\text{balance}(sender, reverted_state) = \text{balance}(sender, state) - \text{gas_fees}(t))$$

Execution Logic:

$$\text{ExecutionResult} = \quad (20)$$

$$\begin{cases} \text{AssertionFailure} & \text{if } \neg solvency_condition \\ (\text{Success}, new_state) & \text{if } solvency_condition \wedge \neg reserve_dipped \\ (\text{Revert}, reverted_state) & \text{if } solvency_condition \wedge reserve_dipped \end{cases} \quad (21)$$

Note: Execute is the black box EVM execution without reserve balance, and returns the state after execution and a map with keys as accounts whose balances were changed during execution and values as the old balances before execution.

Algorithm 3: Reserve Balance Dip Detection

Given:

- Original balances map $original_balances : Address \rightarrow Balance$
- New state new_state
- Transaction t
- $is_emptying$: Is sender allowed to dip into reserve

Define:

$$DippedIntoReserve(*original_balances*, *new_state*, *t*, *is_emptying*) := \tag{22}$$

$$\exists(account, orig_bal) \in original_balances. account \text{ is not SC in } new_state \wedge \tag{23}$$

$$AccountReserveViolated(account, orig_bal, new_state, t, is_emptying) \tag{24}$$

Where:

$$AccountReserveViolated(*account*, *orig_bal*, *new_state*, *t*, *is_emptying*) := \tag{25}$$

$$\text{let } reserve = \min(\text{user_reserve_balance}(account), orig_bal) \tag{26}$$

$$\text{let } current_balance = \text{balance}(account, new_state) \tag{27}$$

$$\text{let } violation_threshold = \begin{cases} \max(reserve - \text{gas_fees}(t), 0) & \text{if } account = t.sender \\ reserve & \text{if } account \neq t.sender \end{cases} \tag{28}$$

($\max(reserve - \text{gas_fees}(t), 0)$ is done to work with the uint datatypes of EVM.)

$$\text{in } ((account \neq t.sender) \vee \neg is_emptying) \wedge (current_balance < violation_threshold) \tag{29}$$

Algorithm 4: Emptying Transaction Check

Given:

- Transaction t
- all_tx : Contains all transactions from block $starting_block$ (inclusive) up to t (inclusive)
- Starting block $starting_block$
- Delegation status $delegated_in_state$

Logic: EOA account (no potential delegation) and no prior transactions in the past k blocks (includes t 's block).

Define:

$$IsEmptying(t, all_tx, starting_block, delegated_in_state) := \quad (30)$$

$$\text{let } prior_txs = \{tx \in all_tx : tx \preceq t \wedge \quad (31)$$

$$\text{blocknumber}(tx) \geq starting_block\} \quad (32)$$

$$\text{let } delegation_condition = delegated_in_state \quad (33)$$

$$\text{let } auth_condition = \exists tx \in prior_txs. HasDelegationAuth(tx, t.sender) \quad (34)$$

$$\text{let } prior_sender_condition = \exists tx \in prior_txs. (PriorSenderTx(tx, t)) \quad (35)$$

Where:

$$PriorSenderTx(tx', t) := tx' \prec t \wedge tx'.sender = t.sender \quad (36)$$

$$\wedge \text{blocknumber}(tx') \geq \text{blocknumber}(t) - k + 1 \quad (37)$$

$$HasDelegationAuth(tx, account) := \text{whether } tx \text{ contains valid (including wrt nonce)} \quad (38)$$

$$\text{delegation authorization from } account \quad (39)$$

Note: Delegation authorization towards null address (i.e., undelegating) in prior transactions also counts in HasDelegationAuth.

Result:

$$IsEmptying(t, all_tx, starting_block, delegated_in_state) \iff \quad (40)$$

$$\neg delegation_condition \wedge \neg auth_condition \quad (41)$$

$$\wedge \neg prior_sender_condition \quad (42)$$

7 Dynamic Base Fee Reference Implementation

```
def fake_exponential(factor: np.uint64, num: np.int64, denom: np.uint64) ->
↳ np.uint64:
    i = Uint256(1)
    output = Int256(0)
    num_accum = Int256( Uint256(factor) * Uint256(denom) )
    while num_accum != 0:
        output += num_accum
        num_accum = (num_accum * Int256(num)) // Int256(Uint256(denom) * i)
        i += Uint256(1)
    return np.uint64(output // Int256(denom))

class DynamicTFM:
    def __init__(self, gas_limit, target_gas, min_base_fee,
                 beta, epsilon, inv_step):
        # protocol params
        self.gas_limit = np.uint64(gas_limit) # 200_000_000
        self.target_gas = np.uint64(target_gas) # 160_000_000
        self.min_base_fee = np.uint64(min_base_fee) # 100_000_000_000
        self.beta = np.uint64(beta) # 96
        self.epsilon = np.int64(epsilon) # 160_000_000
        self.inv_step = np.uint64(inv_step) # 28

        self.base_fee = np.uint64(0)
        self.moment = np.uint64(0)
        self.trend = np.int64(0)

    def update(self, gas_used: np.int64):
        delta = np.int64(gas_used) - self.target_gas
        num = self.epsilon * delta
        denom = self.inv_step * (np.uint64(math.isqrt( np.uint64(max(0,
↳ np.int64(self.moment) - self.trend*self.trend))) ) + self.epsilon) *
↳ (self.gas_limit - self.target_gas)

        self.base_fee = fake_exponential(self.base_fee, num, denom)
        self.base_fee = max(self.base_fee, self.min_base_fee)
        self.trend = ( np.int64(self.beta) * self.trend +
↳ np.int64(np.uint64(100)-self.beta) * (-delta) ) // 100
        self.moment = np.uint64( ( Uint128(self.beta) * Uint128(self.moment) +
↳ (Uint128(100)-Uint128(self.beta)) * Uint128(Int128(delta) *
↳ Int128(delta)) ) // Uint128(100) )
```

Listing 1: Python reference implementation of base fee update rule with integer math