Integrating EVM Networks With Chainlink Services

Before an EVM blockchain network can integrate with Chainlink, it must meet certain technical requirements. These requirements are critical for Chainlink nodes and Chainlink services to function correctly on a given network.

Disclaimer

The standard EVM requirements required to integrate EVM blockchain networks with Chainlink services can vary, are subject to change, and are provided here for reference purposes only. Chainlink services may have unique requirements that are in addition to the requirements discussed herein.

Standard EVM requirements

Solidity global variables and opcode implementation

Solidity global variables and opcode implementation constructs must meet the following requirements in order for Chainlink services to operate correctly and as expected:

- Global variables:Support all global variables as specified in the SolidityBlock and Transaction Properties documentation. For example, the following variables must be supported:* block.blockhashmust return the hash of the requested block for the last 256 blocks
- block.numbermust return the respective chain's block number
- block.chainIDmust return the current chain ID
- block.timestampmust return the current block timestamp as seconds since unix epoch
- Opcodes:Support all opcodes and expected behaviors from theOpCodes.sol contract
- Precompiles:Must support all precompile contracts and expected behaviors listed in the SolidityMathematical and Cryptographic Functions documentation
- Contract size:Must support the maximum contract size defined inEIP-170
- Nonce:The transaction nonce must increase as transactions are confirmed and propagated to all nodes in the network.

Finality

Blockchain development teams must ensure that blocks with a commitment level offinalizedare actually final. The properties of the finality mechanism, including underlying assumptions and conditions under which finality violations could occur, must be clearly documented and communicated to application developers in the blockchain ecosystem.

Furthermore, this information should be accessible through RPC API tagsfinalizedfrom theJSON-RPC specification tags described later in this document.

Standardized RPCs with SLAs

Chainlink nodes use RPCs to communicate with the chain and perform soak testing. It is not possible to ensure the functionality of Chainlink services if RPCs are unstable, underperforming, or nonexistent. RPCs must meet the following requirements:

Dedicated RPC node:

- The chain must provide instructions and hardware requirements to set up and run a full node.
- The archive node setup must also be provided and allow queries of blocks from genesis with transaction history and logs.
- The RPC node must enable and allow configurable settings for the following items:* Batch calls
- Log lookbacks
- HTTPS and WSS connections

RPC providers:

- Three separate independent RPC providers must be available.
- RPC providers must ensure there is no rate limit.
- RPC providers must have a valid SSL certificate.
- During the trailing 30 days, the RPC providers must meet the following RPC performance requirements:* Uptime: At least 99.9%
- Throughput: Support at least 300 calls per second
- Latency: Less than 250ms
- Support SLA: For SEV1 issues, provide a Time to Answer (TTA) of at most 1 hour

Support the Ethereum JSON-RPC Specification

The chain must support theEthereum JSON-RPC Specification . Chainlink services use several methods to operate on the

chain and require a specific response format to those calls in line with the JSON RPC standard of Ethereum. If a response does not match this required format, the call fails and the Chainlink node will stop functioning properly.

The following methods are specifically required and must follow theEthereum RPC API specification :

- GetCode
- Call
- ChainID
- SendTransaction
- SendRawTransaction
- GetTransactionReceipt
- GetTransactionByHash
- EstimateGas
- GasPrice * Must accept theblockhashparam as defined inEIP-234
- GetTransactionCount
- GetLogs * Must follow the spec as defined inEIP-1474 . The "latest" block number returned byGetBlockByNumbermust also be served byGetLogswith logs.
- GetBalance
- GetBlockByNumber
- GetBlockByHash

The above RPC methods must have the expected request and response params with expected data types and values as described in theExecution-api spec andEthereum RPC API Spec .

The network must also support the following items:

- Subscription Methods:Websocket JSON-RPC subscription methods * eth_subscribewith support for subscription tonewHeadsandlogs
- Tags:The RPC methods must support thefinalized,latest, andpendingtags where applicable. They must also support natural numbers for blocks.
- Batch Requests:Must support batching of requests for theGetLogsandGetBlockByNumbermethods.
- Response size:Any RPC request including the batch requests must be within the allowed size limit of around 173MB.

eth_sendRawTransactionerror message mapping to Geth client error messages

Chains must provide an error message mapping between their specific implementation to the error messages detailed below.

When theeth_sendRawTransactioncall fails, Chainlink nodes must be able to recognize these error categories and determine the next appropriate action. If the error categories are different or cannot be mapped correctly, the Chainlink node will stop functioning properly and stop sending transactions to the chain. The following error messages are specifically critical:

ErrorDescriptionNonceTooLowReturned when the nonce used for the transaction is too low to use. This nonce likely has been already used on the chain previously.NonceTooHighReturned when the nonce used for the transaction is higher than what the chain can use right now.ReplacementTransactionUnderpricedReturned when the transaction gas price used is too low. There is another transaction with the same nonce in the queue, with a higher price.LimitReachedReturned when there are too many outstanding transactions on the node.TransactionAlreadyInMempoolReturned when this current transaction was already received and stored.TerminallyUnderpricedReturned when the transaction's gas price is too low and won't be accepted by the node.InsufficientEthReturned when the account doesn't have enough funds to send this transaction.TxFeeExceedsCapReturned when the transaction gas fees exceed the configured cap by this node, and won't be accepted.L2FeeTooLowSpecific for Ethereum L2s only. Returned when the gas fees are too low, When this error occurs the Suggested Gas Price is fetched again transaction is retried.L2FeeTooHighSpecific for Ethereum L2s only. Returned when the total fee is too high. When this error occurs the Suggested Gas Price is fetched again transaction is retried.L2FullSpecific for Ethereum L2s only. The node is too full, and cannot handle more transactions.TransactionAlreadyMinedReturned when the current transaction was already accepted and mined into a block.FatalReturn when something is seriously wrong with the transaction, and the transaction will never be accepted in the current format. For examples of how other chains or clients are using these categories, see theerror.go file in thego-ethereum repo on GitHub.

For chains with zk-proofs, chains must reject transactions that cause zk-proof overflow with a uniquely identifiable error message.

Any other reasons why transactions might be rejected by a node or sequencer other than malformed input/gasLimits must be detailed.

Clarify use of transaction types

Fortransaction types other than0x0 - Legacy,0x1 - Access List,0x2 - Dynamic, and0x3 - Blob, networks must clarify how each transaction type is used. Chainlink nodes must know if the chain uses other types for regular transactions with regular

gas so it can correctly estimate gas costs.

Multi-signature wallet support

The chain must provide a supported and audited multi-signature wallet implementation with a UI.

Block explorer support

The chain must provide a block explorer and support for contract and verification APIs.