Troubleshooting: Building Arbitrum dApps

{"@context":"https://schema.org","@type":"FAQPage","mainEntity":[{"@type":"Question","name":"How does gas work on Arbitrum?","acceptedAnswer":{"@type":"Answer","text":"

Fees on Arbitrum chains are collected on L2 in the chains' native currency (ETH on both Arbitrum One and Nova).

\n\n

A transaction fee is comprised of both an L1 and an L2 component:

\n\n

The L1 component is meant to compensate the Sequencer for the cost of posting transactions on L1 (but no more). (See L1 Pricing.)

\n\n

The L2 component covers the cost of operating the L2 chain; it uses Geth for gas calculation and thus behaves nearly identically to L1 Ethereum. One difference is that unlike on Ethereum, Arbitrum chains enforce a gas price floor; currently 0.1 gwei on Arbitrum One and 0.01 gwei on Nova (See Gas).

\n\n

L2 Gas price adjusts responsively to chain congestion, ala EIP 1559.

\n\n

\n\n"}},{"@type":"Question","name":"I tried to create a retryable ticket but the transaction reverted on L1. How can I debug the issue?","acceptedAnswer":{"@type":"Answer","text":"

Creation of retryable tickets can revert with one of these custom errors:

\n\n

1. **InsufficientValue**: not enough gas included in your L1 transaction's callvalue to cover the total cost of your retryable ticket; i.e., `msg.value < (maxSubmissionCost + l2CallValue + gasLimit * maxFeePerGas)`. Note that your L1 transaction's callvalue must cover this full cost. See Retryable Tickets Lifecycle for more information.
   \n
2. **InsufficientSubmissionCost:** provided submission cost isn't high enough to create your retryable ticket.
   \n
3. **GasLimitTooLarge:** provided gas limit is greater than 2^64
   \n
4. **DataTooLarge**: provided data is greater than 117.964 KB (90% of Geth's 128 KB transaction size limit).
   \n

\n

\n\n

To figure out which error caused your transaction to revert, we recommend using etherscan's Parity VM trace support (Tenderly is generally a very useful debugging tool; however, it can be buggy when it comes to custom Geth errors).

\n\n

Use the following link to view the Parity VM trace of your failed transaction (replacing the tx-hash with your own, and using the appropriate etherscan root url):

\n\n

https://etherscan.io/vmtrace?txhash=0x51a8088c9b319bbad649c36d9cf2b4e9b61a6099a158181676c8e79dbce2df58&type=parity#raw

\n\n

To find out the reversion error signature, go to the \"Raw Traces\" tab, and scroll down to find the last \"subtrace\" in which your transaction is reverted. Then find \"output\" field of that subtrace.

\n\n

(In the above example the desirable \"output\" is:
\n{'0xfadf238a0000000000000000000000000000000000000000000000000000000000000000c4df7e2903b0000000000000000000000000000000000000000000000000000000000000a39a1d002808'})

\n\n

\n\n

The first four bytes of the output is the custom error signature;in our example it's `0xfadf238a` .

\n\n

\nTo let's find out which is custom error this signature represents, we can use this handy tool by Samzcsun: https://sig.eth.samczsun.com/

\n\n

Checking `0xfadf238a` gives us `InsufficientSubmissionCost(uint256,uint256)`.
\n

\n\n

\n\n"}},{"@type":"Question","name":"How is the L1 portion of an Arbitrum transaction's gas fee computed? ","acceptedAnswer":{"@type":"Answer","text":"

The L1 fee that a transaction is required to pay is determined by compressing its data with brotli and multiplying the size of the result (in bytes) by ArbOS's current calldata price; the latter value can be queried via the `getPricesInWei` method of the `ArbGasInfo` precompile. You can find more information about gas calculations in Understanding Arbitrum: 2-Dimensional Fees and How to estimate gas in Arbitrum

\n\n

\n\n"}},{"@type":"Question","name":"What is a retryable ticket's \"submission fee\"? How can I calculate it? What happens if I the fee I provide is insufficient?","acceptedAnswer":{"@type":"Answer","text":"

A retryable's submission fee is a special fee a user must pay to create a retryable ticket. The fee is directly proportional to the size of the L1 calldata the retryable ticket uses. The fee can be queried using the `Inbox.calculateRetryableSubmissionFee` method. If insufficient fee is provided, the transaction will revert on L1, and the ticket won't get created.

\n\n

\n\n"}},{"@type":"Question","name":"Which method in the Inbox contract should I use to submit a retryable ticket (aka L1 to L2 message)?","acceptedAnswer":{"@type":"Answer","text":"

The method you should (almost certainly) use is `Inbox.createRetryableTicket`. There is an alternative method, `Inbox.unsafeCreateRetryableTicket`, which, as the name suggests, should only be used by those who fully understand its implications.

\n\n

There are two differences between `createRetryableTicket` and `unsafeCreateRetryableTicket`:

\n\n

1. Method `createRetryableTicket` will check that provided L1 callvalue is sufficient to cover the costs of creating and executing the retryable ticket (at the specified parameters) and otherwise revert directly at L1. `unsafeCreateRetryableTicket`, in contrast, will allow a retryable ticket to be created that is guaranteed to revert on L2.
   \n
2. Method `createRetryableTicket` will check if either the provided `excessFeeRefundAddress` or the `callValueRefundAddress` are contracts on L1; if they are, to prevent the situation where refunds are *guaranteed* to be irrecoverable on L2, it will convert them to their address alias, providing a *potential* path for fund recovery. `unsafeCreateRetryableTicket` will allow the creation of a retryable ticket with refund addresses that are L1 contracts; since no L1 contract can alias to an address that is also itself an L1 contract, refunds to these addresses on L2 will be irrecoverable.
   \n

\n

(Astute observers may note a third ticket creation method, `createRetryableTicketNoRefundAliasRewrite`; this is included only for backwards compatibility, but should be considered deprecated in favor of `unsafeCreateRetryableTicket`)

\n\n"}},{"@type":"Question","name":"Why do I get \"custom tx type\" errors when I use hardhat?","acceptedAnswer":{"@type":"Answer","text":"

In Arbitrum, we use a number of non-standard EIP-2718 typed transactions. See here for the full list and the rationale.

\n\n

Note that if you're using Hardhat, v2.12.2 added support for forking networks like Arbitrum with custom transaction types (find more information here).

\n\n

\n\n"}},{"@type":"Question","name":"Why does it look like two identical transactions consume a different amount of gas?","acceptedAnswer":{"@type":"Answer","text":"

\n\n

Calling an Arbitrum node's `eth_estimateGas` RPC returns a value sufficient to cover both the L1 and L2 components of the fee for the current gas price; this is the value that, e.g., will appear in users' wallets in the \"gas limit\" field.

\n\n

Thus, if the L1 calldata price changes over time, it will appear (in e.g., a wallet) that a transaction's gas limit is changing. In fact, the L2 gas limit isn't changing, merely the total gas required to cover the transaction's L1 + L2 fees.

\n\n

See 2-D fees and How to estimate gas in Arbitrum for more.

\n\n

\n\n

\n\n"}},{"@type":"Question","name":"Why am I getting error \"429 Too Many Requests\" when using one of Offchain Labs' Public RPCs?","acceptedAnswer":{"@type":"Answer","text":"

Offchain Labs offers public RPCs for free, but limits requests to prevent DOSing. Hitting the rate limit could come from your request frequency and/or the resources required to process the requests. If you are hitting our rate limit, we recommend running your own node or using a third party node provider.

\n\n

\n\n"}},{"@type":"Question","name":"How do block.number and block.timestamp work on Arbitrum?","acceptedAnswer":{"@type":"Answer","text":"

Solidity calls to `block.number` on Arbitrum will return the block number/ timestamp of the underlying L1 on a slight delay; i.e., updated every few minutes. Note that L2 block numbers (i.e., as seen in block explorers / returned by RPCs) are different, and are typically updated roughly every second.

\n\n

Solidity calls to `block.timestamp` on Arbitrum are not linked to the timestamp of the L1 block. It is updated every L2 block based on the sequencer's clock. Furthermore, for transactions that are force-included from L1 (bypassing the Sequencer) `block.timestamp` will be equal to the L1 timestamp when the transaction was put in the delayed inbox on L1 (not force-included), or the L2 timestamp of the previous L2 block (whichever is greater of the two timestamps).

\n\n

For more info, see block numbers and time.

\n\n

\n\n"}},{"@type":"Question","name":"Do I need to download any special npm libraries in order to use web3.js, ethers.js or viem on Arbitrum? ","acceptedAnswer":{"@type":"Answer","text":"

Nope, web3.js, ethers.js and viem will work out of the box just like they do on L1 Ethereum.

\n\n

Once upon a time, Arbitrum developers were required to download supplemental packages with names like \"arb-provider-ethers\" and \"arb-ethers-web3-bridge\", but these packages are deprecated and no longer required! Any guide that directs devs to use them should be considered outdated.

\n\n

\n\n

\n\n"}},{"@type":"Question","name":"How many block numbers must we wait for in Arbitrum before we can confidently state that the transaction has reached finality?","acceptedAnswer":{"@type":"Answer","text":"

Arbitrum's block intervals fluctuate with throughput, so relying on block numbers for finality isn't recommended. However, Arbitrum nodes support Ethereum's JSON RPC, enabling the use of eth_getBlockByNumber() to determine block finality. Here, we provide additional details on how to achieve this.

\n\n

\n\n

You can use eth_getBlockByNumber() with the string \"latest\", \"safe\", or \"finalized\", each offering varying degrees of finality:

\n\n

\n\n

- latest: Provides you with the most recent Arbitrum block number that the node has observed on the L1 and indicates that the Sequencer's batch has been just published as an L1 block on the Ethereum network. It's important to note that this block has the potential to be re-orged but you can consider and trust this block as final, if you trust the sequencer.
  \n
- safe: Provides you with the most recent Arbitrum block number that has achieved attestations from a two-thirds majority of Ethereum's validator set. This occurs when the Sequencer's batch is posted as an L1 block on Ethereum and then the batch transactions achieve safe finality there. While safe blocks are typically resistant to re-orgs, they can still be re-orged in the event of a significant L1 re-org.
  \n
- finalized: Provides you with the most recent Arbitrum block number that is finalized on Ethereum. This means that the Sequencer's batch has been published as an L1 block on the Ethereum network and has reached a substantial depth, making it eligible for hard finality. Unlike safe blocks, finalized blocks are highly improbable to undergo re-orgs.
  \n

\n

\n\n

To learn more about the different phases of an Arbitrum transaction, from client initiation to Layer 1 confirmation, check out The Lifecycle of an Arbitrum Transaction.

\n\n

\n\n"}},{"@type":"Question","name":"How can I list my token on the Arbitrum Bridge?","acceptedAnswer":{"@type":"Answer","text":"

The L2 token list used in the Arbitrum bridge is generated from the L1 tokens that are part of the token list of Uniswap, Gemini , Coinmarketcap or Coingecko. This is valid for L1-native tokens that have been bridged over to L2, and for L2-native tokens that have been bridged over to L1 as long as they are part of any of those lists.

\n\n

Currently, there isn't any L2-only token list.

\n\n

\n\n

\n\n"}},{"@type":"Question","name":"What is a testnet or a devnet?","acceptedAnswer":{"@type":"Answer","text":"

Testnets (or devnets) primarily serve developers who want to test out the applications they're building without having to use any real mainnet funds.

\n\n

Arbitrum Sepolia is a testnet that has the same full feature-set as the mainnet network. It is also a \"true\" L2 that runs on top of the Sepolia testnet (L1), using it for security and settlement.

\n\n

Users can bridge any asset from the Sepolia testnet (L1) into the Arbitrum Sepolia testnet (and back!), using the official bridge.

\n\n

\n\n"}},{"@type":"Question","name":"Is there any testnet available on Arbitrum?","acceptedAnswer":{"@type":"Answer","text":"

Yes, there's an Arbitrum Sepolia testnet (421614) that uses the Nitro tech stack and runs on top of Ethereum Sepolia. You can find more information here.

\n\n

\n\n"}},{"@type":"Question","name":"When was Arbitrum One upgraded from Classic to Nitro?","acceptedAnswer":{"@type":"Answer","text":"

Arbitrum One was upgraded on August 31st, 2022 (block 22207818), from the Classic stack to the improved Nitro tech stack, maintaining the same state.

\n\n

\n\n"}},{"@type":"Question","name":"Do Arbitrum chains support precompiles that are present on Ethereum?","acceptedAnswer":{"@type":"Answer","text":"

Yes, all Arbitrum chains support all precompiles that Ethereum supports, as well as others that are not present on Ethereum. Check the precompiles reference page for more information about Arbitrum specific precompiles.

\n\n

\n\n"}},{"@type":"Question","name":"What's the contract code size limit in Arbitrum chains?","acceptedAnswer":{"@type":"Answer","text":"

As specified in EIP-170, contracts of up to 24KB are deployable on Arbitrum chains.

\n\n

\n\n"}},{"@type":"Question","name":"How can I find the L2 block(s) that corresponds to a given L1 block?","acceptedAnswer":{"@type":"Answer","text":"

First of all, you should be familiar with how block numbers behave on Arbitrum. You can find information about it in Block numbers and time.

\n\n

When you query an RPC node for a transaction receipt or a block information, you obtain as part of the result the property l1BlockNumber, which is the L1 block number that the sequencer viewed when it processed the transaction.

\n\n

With that, although it might be computationally complex, you can binary search the L1 block number you are looking for, and get all L2 blocks that have that l1BlockNumber.

\n\n

If you want a more specific result, you can perform the same operation with the timestamp from the L1 block, instead of the actual block number.

\n\n"}},{"@type":"Question","name":"Why do some old transactions have extremely high gas prices when querying them?","acceptedAnswer":{"@type":"Answer","text":"

When Arbitrum One was running under the Arbitrum Classic stack (before Nitro), the gas price was an unbounded bid, so when requesting those transactions via RPC, you may obtain a very high amount in the `gasPrice` property.

\n\n

Instead of that, it is recommended to look at the `effectiveGasPrice` property from the transaction receipt.

\n\n

\n\n"}},{"@type":"Question","name":"What is the WASM module root?","acceptedAnswer":{"@type":"Answer","text":"

The WASM module root is a 32 byte hash, which is a merkelization of the Go replay binary and its dependencies.

\n\n

The replay binary is much too large to post on-chain, so this hash is set in the L1 rollup contract to determine the correct replay binary during fraud proofs.

\n\n

You can find more information in How to customize your Orbit chain's behavior.

\n\n

\n\n"}},{"@type":"Question","name":"Why do I get a \"gas required exceeds allowance\" when trying to estimate the gas costs of a request?","acceptedAnswer":{"@type":"Answer","text":"

During an `eth_estimateGas` call the actual request will be simulated on the node, so if the transaction reverts or if there aren't enough funds in the wallet that's making the call (usually the `from` parameter), the eth_estimateGas request will return the error `gas required exceeds allowance`.

\n\n

Make sure you have enough funds in your wallet, and the gas fields of the request (if you're using them) are correctly set.

\n\n"}},{"@type":"Question","name":"How can I verify that an L2 block has been processed as part of a specific RBlock?","acceptedAnswer":{"@type":"Answer","text":"

If you want to verify that the latest confirmed (or created) assertion has processed a specific L2 block, you can follow these steps:

\n\n

1. From the rollup contract, obtain the latest confirmed (or created) Rblock through the function `latestConfirmed` (or `latestNodeCreated`). In this context, we refer to RBlocks as \"nodes\".
   \n
2. Obtain the node information through `getNode`
   \n
3. Find the `NodeCreated` event that was emitted when that node was created.
   \n
4. In that NodeCreated event, there's an `assertion` property that contains the state of the chain before processing the specified blocks, and after processing them. Get the `afterState.globalState` property
   \n
5. That value contains a bytes32Vals array with the latest L2 block hash processed in the first element.
   \n

\n

You can find an example script in our arbitrum-tutorials repository.

\n\n"}},{"@type":"Question","name":"Why is the fee of some Classic transactions slightly different than the multiplication of gasLimit and effectiveGasPrice?","acceptedAnswer":{"@type":"Answer","text":"

Gas prices in Classic transactions worked a bit differently than in Nitro transactions. Classic transactions handled four different prices: L1 fixed, L1 calldata, L2 computation and L2 storage. You can see all those prices in the Advanced TxInfo of Arbiscan (here's an example).

\n\n

When querying the receipt of a Classic transaction on a Nitro node, there's some calculation done to get an effectiveGasPrice that is close to (but not exactly) what those four prices represent. That's why if you multiply the gasLimit by the effectiveGasPrice you might end up with a transaction fee that is slightly different than the actual fee paid.

\n\n

To get the exact fees paid, you can query a Classic node, which will return all the exact information.

\n\n

\n\n"}},{"@type":"Question","name":"How can I update the information of my bridged token on Arbiscan?","acceptedAnswer":{"@type":"Answer","text":"

If you have a native-L1 token that was bridged to L2 via the standard gateway, you might find that you can't claim ownership of the L2 contract of your token as it was generated by another contract.

\n\n

To update its information on Arbiscan (logo, socials, etc.), you can open a ticket through Arbiscan support system and request them to replicate the information of your token on L1 to L2.

\n\n

\n\n"}},{"@type":"Question","name":"Why does my transaction revert with InvalidFEOpcode when using Foundry?","acceptedAnswer":{"@type":"Answer","text":"

Foundry and other similar development tools that enable chain forking, do not support Arbitrum precompiles. If your transaction is calling a precompile, it is likely that it will revert with `InvalidFEOpcode`.

\n\n

To rule out that possibility, it is recommended to send the transaction with a different tool.

\n\n

\n\n"}},{"@type":"Question","name":"Why do I receive an \"intrinsic gas too low\" error when sending a transaction even with a high gas price?","acceptedAnswer":{"@type":"Answer","text":"

The error intrinsic gas too low usually refers to not providing enough gas to pay for the L1 component of the transaction fees. This is usually a problem related to not setting a high enough gas limit (instead of gas price), due to how Arbitrum handles gas. You can find more information in the article Understanding Arbitrum: 2-dimensional fees and the page How to estimate gas.

\n\n

\n\n"}}]}

How does gas work on Arbitrum?

Fees on Arbitrum chains are collected on L2 in the chains' native currency (ETH on both Arbitrum One and Nova).

A transaction fee is comprised of both an L1 and an L2 component:

The L1 component is meant to compensate the Sequencer for the cost of posting transactions on L1 (but no more). (SeeL1 Pricing .)

The L2 component covers the cost of operating the L2 chain; it uses Geth for gas calculation and thus behaves nearly identically to L1 Ethereum. One difference is that unlike on Ethereum, Arbitrum chains enforce a gas price floor; currently 0.1 gwei on Arbitrum One and 0.01 gwei on Nova (SeeGas).

L2 Gas price adjusts responsively to chain congestion, ala EIP 1559.

I tried to create a retryable ticket but the transaction reverted on L1. How can I debug the issue?

Creation of retryable tickets can revert with one of these custom errors:

1. InsufficientValue
2. : not enough gas included in your L1 transaction's callvalue to cover the total cost of your retryable ticket; i.e.,msg.value < (maxSubmissionCost + l2CallValue + gasLimit * maxFeePerGas)
3. . Note that your L1 transaction's callvalue must cover this full cost. SeeRetryable Tickets Lifecycle
4. for more information.
5. InsufficientSubmissionCost:
6. provided submission cost isn't high enough to create your retryable ticket.
7. GasLimitTooLarge:
8. provided gas limit is greater than 2^64
9. DataTooLarge
10. : provided data is greater than 117.964 KB (90% of Geth's 128 KB transaction size limit).

To figure out which error caused your transaction to revert, we recommend using etherscan's Parity VM trace support (Tenderly is generally a very useful debugging tool; however, it can be buggy when it comes to custom Geth errors).

Use the following link to view the Parity VM trace of your failed transaction (replacing the tx-hash with your own, and using the appropriate etherscan root url):

https://etherscan.io/vmtrace?txhash=0x51a8088c9b319bbad649c36d9cf2b4e9b61a6099a158181676c8e79dbce2df58&type=parity#raw

To find out the reversion error signature, go to the "Raw Traces" tab, and scroll down to find the last "subtrace" in which your transaction is reverted. Then find "output" field of that subtrace.

(In the above example the desirable "output" is:
{'0xfadf238a0000000000000000000000000000000000000000000000000000000000000000c4df7e2903b0000000000000000000000000000000000000000000000000000000000000000a39a1d002808'}
)

The first four bytes of the output is the custom error signature; in our example it's0xfadf238a .

To let's find out which is custom error this signature represents, we can use this handy tool by Samzcsunhttps://sig.eth.samczsun.com/

Checking0xfadf238a gives usInsufficientSubmissionCost(uint256,uint256) .

How is the L1 portion of an Arbitrum transaction's gas fee computed?

The L1 fee that a transaction is required to pay is determined by compressing its data with brotli and multiplying the size of the result (in bytes) by ArbOS's current calldata price; the latter value can be queried via thegetPricesInWei method of theArbGasInfo precompile. You can find more information about gas calculations inUnderstanding Arbitrum: 2-Dimensional Fees andHow to estimate gas in Arbitrum.

What is a retryable ticket's "submission fee"? How can I calculate it? What happens if I the fee I provide is insufficient?

Aretryable's submission fee is a special fee a user must pay to create a retryable ticket. The fee is directly proportional to the size of the L1 calldata the retryable ticket uses. The fee can be queried using theInbox.calculateRetryableSubmissionFee method. If insufficient fee is provided, the transaction will revert on L1, and the ticket won't get created.

Which method in the Inbox contract should I use to submit a retryable ticket (aka L1 to L2 message)?

The method you should (almost certainly) use isInbox.createRetryableTicket . There is an alternative method,Inbox.unsafeCreateRetryableTicket , which, as the name suggests, should only be used by those who fully understand its implications.

There are two differences betweencreateRetryableTicket andunsafeCreateRetryableTicket :

1. MethodcreateRetryableTicket
2. will check that provided L1 callvalue is sufficient to cover the costs of creating and executing the retryable ticket (at the specified parameters) and otherwiserevert directly at L1
3. .unsafeCreateRetryableTicket
4. , in contrast, will allow a retryable ticket to be created that is guaranteed to revert on L2.
5. MethodcreateRetryableTicket
6. will check if either the providedexcessFeeRefundAddress
7. or thecallValueRefundAddress
8. are contracts on L1; if they are, to prevent the situation where refunds areguaranteed
9. to be irrecoverable on L2, it will convert them to theiraddress alias
10. , providing apotential
11. path for fund recovery.unsafeCreateRetryableTicket
12. will allow the creation of a retryable ticket with refund addresses that are L1 contracts; since no L1 contract can alias to an address that is also itself an L1 contract, refunds to these addresses on L2 will be irrecoverable.

(Astute observers may note a third ticket creation method,createRetryableTicketNoRefundAliasRewrite ; this is included only for backwards compatibility, but should be considered deprecated in favor ofunsafeCreateRetryableTicket)

Why do I get "custom tx type" errors when I use hardhat?

In Arbitrum, we use a number of non-standard EIP-2718 typed transactions. See here for the full list and the rationale.

Note that if you're using Hardhat, v2.12.2 added support for forking networks like Arbitrum with custom transaction types (find more information here).

Why does it look like two identical transactions consume a different amount of gas?

Calling an Arbitrum node'seth_estimateGas RPC returns a value sufficient to cover both the L1 and L2 components of the fee for the current gas price; this is the value that, e.g., will appear in users' wallets in the "gas limit" field.

Thus, if the L1 calldata price changes over time, it will appear (in e.g., a wallet) that a transaction's gas limit is changing. In fact, the L2 gas limit isn't changing, merely the total gas required to cover the transaction's L1 + L2 fees.

See 2-D fees and How to estimate gas in Arbitrum for more.

Why am I getting error "429 Too Many Requests" when using one of Offchain Labs' Public RPCs?

Offchain Labs offers public RPCs for free, but limits requests to prevent DOSing. Hitting the rate limit could come from your request frequency and/or the resources required to process the requests. If you are hitting our rate limit, we recommend running your own node or using a third party node provider .

How do block.number and block.timestamp work on Arbitrum?

Solidity calls toblock.number on Arbitrum will return the block number/ timestamp of the underlying L1 on a slight delay; i.e., updated every few minutes. Note that L2 block numbers (i.e., as seen in block explorers / returned by RPCs) are different, and are typically updated roughly every second.

Solidity calls toblock.timestamp on Arbitrum are not linked to the timestamp of the L1 block. It is updated every L2 block based on the sequencer's clock. Furthermore, for transactions that are force-included from L1 (bypassing the Sequencer)block.timestamp will be equal to the L1 timestamp when the transaction was put in the delayed inbox on L1 (not force-included), or the L2 timestamp of the previous L2 block (whichever is greater of the two timestamps).

For more info, see block numbers and time .

Do I need to download any special npm libraries in order to use web3.js, ethers.js or viem on Arbitrum?

Nope, web3.js, ethers.js and viem will work out of the box just like they do on L1 Ethereum.

Once upon a time, Arbitrum developers were required to download supplemental packages with names like "arb-provider-ethers" and "arb-ethers-web3-bridge", but these packages are deprecated and no longer required! Any guide that directs devs to use them should be considered outdated.

How many block numbers must we wait for in Arbitrum before we can confidently state that the transaction has reached finality?

Arbitrum's block intervals fluctuate with throughput, so relying on block numbers for finality isn't recommended. However, Arbitrum nodes support Ethereum's JSON RPC, enabling the use of eth_getBlockByNumber() to determine block finality. Here, we provide additional details on how to achieve this.

You can use eth_getBlockByNumber() with the string "latest" ,"safe" , or "finalized" , each offering varying degrees of finality:

- latest
- : Provides you with the most recent Arbitrum block number that the node has observed on the L1 and indicates that the Sequencer's batch has been just published
- as an L1 block on the Ethereum network. It's important to note that this block has the potential to be re-orged but
- you can consider and trust this block as final, if you trust the sequencer.
- safe
- : Provides you with the most recent Arbitrum block number that has achieved attestations from a two-thirds majority of Ethereum's validator set. This occurs when the Sequencer's batch is posted as an L1 block on Ethereum and then the batch transactions achievesafe
- finality there. While safe blocks are typically resistant to re-orgs, they can still be re-orged in the event of a significant L1 re-org.
- finalized
- : Provides you with the most recent Arbitrum block number that is finalized on Ethereum. This means that the Sequencer's batch has been published
- as an L1 block on the Ethereum network and has reached a substantial depth, making it eligible for hard finality. Unlikesafe
- blocks,finalized
- blocks are highly improbable to undergo re-orgs.

To learn more about the different phases of an Arbitrum transaction, from client initiation to Layer 1 confirmation, check out The Lifecycle of an Arbitrum Transaction .

How can I list my token on the Arbitrum Bridge?

The L2 token list used in the Arbitrum bridge is generated from the L1 tokens that are part of the token list of Uniswap ,Gemini ,Coinmarketcap or Coingecko . This is valid for L1-native tokens that have been bridged over to L2, and for L2-native tokens that have been bridged over to L1 as long as they are part of any of those lists.

Currently, there isn't any L2-only token list.

What is a testnet or a devnet?

Testnets (or devnets) primarily serve developers who want to test out the applications they're building without having to use any real mainnet funds.

Arbitrum Sepolia is a testnet that has the same full feature-set as the mainnet network. It is also a "true" L2 that runs on top of the Sepolia testnet (L1), using it for security and settlement.

Users can bridge any asset from the Sepolia testnet (L1) into the Arbitrum Sepolia testnet (and back!), using the official bridge .

Is there any testnet available on Arbitrum?

Yes, there's an Arbitrum Sepolia testnet (421614) that uses the Nitro tech stack and runs on top of Ethereum Sepolia. You can find more information here .

When was Arbitrum One upgraded from Classic to Nitro?

Arbitrum One was upgraded on August 31st, 2022 (block 22207818), from the Classic stack to the improved Nitro tech stack, maintaining the same state.

Do Arbitrum chains support precompiles that are present on Ethereum?

Yes, all Arbitrum chains support all precompiles that Ethereum supports, as well as others that are not present on Ethereum. Check the precompiles reference page for more information about Arbitrum specific precompiles.

What's the contract code size limit in Arbitrum chains?

As specified in EIP-170 , contracts of up to 24KB are deployable on Arbitrum chains.

How can I find the L2 block(s) that corresponds to a given L1 block?

First of all, you should be familiar with how block numbers behave on Arbitrum. You can find information about it in Block numbers and time .

When you query an RPC node for a transaction receipt or a block information, you obtain as part of the result the propertyl1BlockNumber , which is the L1 block number that the sequencer viewed when it processed the transaction.

With that, although it might be computationally complex, you can binary search the L1 block number you are looking for, and get all L2 blocks that have thatl1BlockNumber .

If you want a more specific result, you can perform the same operation with the timestamp from the L1 block, instead of the actual block number.

Why do some old transactions have extremely high gas prices when querying them?

When Arbitrum One was running under the Arbitrum Classic stack (before Nitro), the gas price was an unbounded bid, so when requesting those transactions via RPC, you may obtain a very high amount in thegasPrice property.

Instead of that, it is recommended to look at theeffectiveGasPrice property from the transaction receipt.

What is the WASM module root?

The WASM module root is a 32 byte hash, which is a merkelization of the Go replay binary and its dependencies.

The replay binary is much too large to post on-chain, so this hash is set in the L1 rollup contract to determine the correct replay binary during fraud proofs.

You can find more information inHow to customize your Orbit chain's behavior .

Why do I get a "gas required exceeds allowance" when trying to estimate the gas costs of a request?

During aneth_estimateGas call the actual request will be simulated on the node, so if the transaction reverts or if there aren't enough funds in the wallet that's making the call (usually thefrom parameter), the eth_estimateGas request will return the errorgas required exceeds allowance .

Make sure you have enough funds in your wallet, and the gas fields of the request (if you're using them) are correctly set.

How can I verify that an L2 block has been processed as part of a specific RBlock?

If you want to verify that the latest confirmed (or created) assertion has processed a specific L2 block, you can follow these steps:

1. From the rollup contract, obtain the latest confirmed (or created) Rblock through the functionlatestConfirmed
2. (orlatestNodeCreated
3.). In this context, we refer to RBlocks as "nodes".
4. Obtain the node information throughgetNode
5. Find theNodeCreated
6. event that was emitted when that node was created.
7. In that NodeCreated event, there's anassertion
8. property that contains the state of the chain before processing the specified blocks, and after processing them. Get theafterState.globalState
9. property
10. That value contains a bytes32Vals array with the latest L2 block hash processed in the first element.

You can find an example script in ourarbitrum-tutorials repository.

Why is the fee of some Classic transactions slightly different than the multiplication of gasLimit and effectiveGasPrice?

Gas prices in Classic transactions worked a bit differently than in Nitro transactions. Classic transactions handled four different prices: L1 fixed, L1 calldata, L2 computation and L2 storage. You can see all those prices in the Advanced TxInfo of Arbiscan (here's anexample).

When querying the receipt of a Classic transaction on a Nitro node, there's some calculation done to get an effectiveGasPrice that is close to (but not exactly) what those four prices represent. That's why if you multiply the gasLimit by the effectiveGasPrice you might end up with a transaction fee that is slightly different than the actual fee paid.

To get the exact fees paid, you can query a Classic node, which will return all the exact information.

How can I update the information of my bridged token on Arbiscan?

If you have a native-L1 token that was bridged to L2 via the standard gateway, you might find that you can't claim ownership of the L2 contract of your token as it was generated by another contract.

To update its information on Arbiscan (logo, socials, etc.), you can open a ticket throughArbiscan support system and request them to replicate the information of your token on L1 to L2.

Why does my transaction revert with InvalidFEOpcode when using Foundry?

Foundry and other similar development tools that enable chain forking, do not support Arbitrum precompiles. If your transaction is calling a precompile, it is likely that it will revert withInvalidFEOpcode .

To rule out that possibility, it is recommended to send the transaction with a different tool.

Why do I receive an "intrinsic gas too low" error when sending a transaction even with a high gas price?

The errorintrinsic gas too low usually refers to not providing enough gas to pay for the L1 component of the transaction fees. This is usually a problem related to not setting a high enough gas limit (instead of gas price), due to how Arbitrum handles gas. You can find more information in the articleUnderstanding Arbitrum: 2-dimensional fees and the pageHow to estimate gas . Last updatedonMar 19, 2024Previous Arbitrum: Understanding the risks Next A gentle introduction: Orbit chains