Why Nitro?

Nitro represents the latest step in the evolution of Arbitrum technology; it is an upgrade from the tech stack first released on the mainnet Arbitrum One chain, which we now refer to as “Arbitrum Classic” (and several steps beyond what was described in theinitial Arbitrum whitepaper back in 2018). Here, we’ll explain the rationale behind the Nitro upgrade, and outline Nitro’s core benefits over the classic system.

Nitro vs. Classic

Viewed from a distance, the Classic and Nitro systems do similar things: both seek to create an execution environment as close to the EVM as possible which operates as a second layer to Ethereum; i.e., safety of the L2 virtual machine’s state updates can be guaranteed and enforced via succinct fraud proofs on Ethereum itself.

In Arbitrum Classic, this was achieved via a custom-made virtual machine, which we call the Arbitrum Virtual Machine (AVM). The implementation of Arbitrum’s L2 state machine—known as“ArbOS” — is effectively a program that is compiled and uploaded to the AVM; ArbOS includes (among other things) the ability to emulate EVM execution.

In Nitro, instead of using the AVM for low-level instructions, we use WebAssembly (Wasm). Since Go code can be compiled down to Wasm, we can implement the ArbOS program in Go, and include within it (as a sub-module)Geth itself , the most widely used Ethereum implementation.

This architecture—in which Geth’s EVM implementation can be used directly—is Nitro’s defining feature, and is principally what we’re talking about when we talk about “Nitro.” Most of Nitro’s benefits are a direct or indirect consequence of this design choice. We can summarize these benefits as follows: lower fees, better Ethereum compatibility, and simplicity.

Lower Fees

(Optimistic)^2 Execution

To understand the core of Nitro’s efficiency, we have to dig a little deeper into the classic AVM. In classic, high-level code (Solidity, Vyper, etc.) would be initially compiled down to the EVM bytecode (as though it were to be deployed on Ethereum). This bytecode would then be transpiled to its corresponding AVM instructions by ArbOS; this AVM bytecode would function both as the instructions for running the L2 VM, and the inputs used to prove fraud; in an interactive fraud proof, two validators dissect a segment of AVM bytecode until a “one step proof” — i.e., a state transition that represents a single AVM opcode — would be executed in the EVM of the L1 itself.

Nitro has a similar bytecode-sandwich-like structure;to prove fraud in Nitro , the node’s Go code is compiled into WebAssembly (Wasm), the individual instructions of which are ultimately similarly dissected over to zero-in on an invalid state update. There is, however, a crucial difference: Nitro, being essentially the EVM, periodically produces Ethereum-esque blocks; we can think of these blocks as natural state-checkpoints within a larger assertion of an L2 state update. Nitro takes advantage of this by splitting the interactive fraud proof game into 2 phases: first, two disputing parties narrow down their disagreement to a single block; then (and only then) do they compile the block to Wasm, and thereby continue to narrow down their dispute to Wasm instruction. Thus, this Wasm compilation step only needs to happen when a dispute occurs.

It’s worth reiterating this distinction: in classic, the code executed in the happy/common case is equivalent to the code used in a fraud proof, whereas in Nitro, we can have different contexts for the two cases for execution and for proving. When a claim is being disputed, we ultimately compile down to Wasm bytecode, but in the happy/common case, we can execute the node’s Go code natively, i.e., in whatever execution environment one’s machine uses. Essentially, Nitro is capable of being even more “optimistic” in its execution, compiling to Wasm only just-in-time as required. The common case of native execution is happily far faster and more performant, and better node performance, of course, translates to lower fees for end users.

Calldata Compression

Typically, the bulk of an Arbitrum Rollup transaction’s fee is covering the cost to post its data on Ethereum. Fundamentally, any rollup must post data on L1 sufficient for reconstruction and validation of the L2 state; beyond that, L2s can be flexible in deciding on what data format to use. Given the relatively high cost of posting data to L1, a natural optimization is to (losslessly) compress data before posting it on L1, and have the L2 environment handle decompressing it.

The flexibility that Arbitrum core architecture offers meant that even in the classic AVM, such decompression could have been implemented in principle. However, given that the AVM was custom-built for Arbitrum, this would have meant building a custom, hand-rolled implementation of a compression algorithm, which, practically speaking, represented a prohibitively high technical risk.

The Nitro architecture, however, fundamentally requires only that its VM can be compiled down to Wasm; so not just Geth, but any Go code can be incorporated. Thus, Nitro can (and does) use widely used, battle-tested compression libraries for calldata compression, and thus significantly reduces the cost of posting transaction batches.

Note that supporting calldata compression also requires a more sophisticated mechanism fordetermining the price of calldata and ensuring that batch posters are ultimately properly compensated, a mechanism which Nitro also introduces.

Closer EVM Compatibility

The classic AVM achieved a strong degree of EVM compatibility with its ability to handle any EVM opcodes. However, being a distinct VM, the AVM's internal behavior in some ways diverged with that of the EVM. Most noticeable for smart contract developers was the denomination of "ArbGas", whose units didn't correspond to Ethereum L1 gas; e.g., a simple transfer takes 21,000 gas on L1 but over 100,000 ArbGas in the AVM. This meant that contracts that included gas calculation logic that were initially built for L1 had to be modified accordingly to be deployed on L2, and likewise with any client-side tooling with similar hardcoded expectations about a chain's gas. With Nitro,gas on L1 and L2 essentially correspond 1:1.

(Note that transactions have to cover the total cost of both L2 execution and L1 calldata; the value returned by Arbitrum nodes'eth_estimateGas RPC — and in turn, the value users will see in their wallets — is calculated to be sufficient to cover this total cost. See2-D fees for more.)

Additionally, node functionality peripheral to execution itself, but still important / expected by much tooling and infrastructure — e.g. support for transaction tracing — is essentially inherited out-of-the-box in Nitro, giving Nitro stronger compatibility with Ethereum not just within its virtual machine, but also with how clients interact with it.

In short, there's no better way to achieve Ethereum compatibility than to reuse the Ethereum software itself.

Simplicity

Having code that is as simple and easy to reason about as possible is important for L2 systems, which are inevitably complex. The classic stack represents a large codebase built in-house, which requires a fair amount of time and overhead to understand. The AVM together with ArbOS effectively constitute a full blockchain protocol built from the ground up. Since the AVM was custom-built, with no high-level languages yet created for it, the ArbOS logic had to be implemented in what was essentially a custom language — called "mini" — along with a mini-to-AVM compiler.

Nitro's direct usage of Geth means most of the work of creating an L2 VM is inherited right out of the box. The ArbOS custom logic (which, happily, can now be written in Go instead of mini), is much slimmer than in the classic stack; since the work of emulating the EVM is now handled by the Geth software, ArbOS needs only to implement the things specific and necessary for layer 2 (i.e., L1/L2 gas accounting, special message types for cross-chain transactions, etc.) Leaner, simpler code — much of which directly inherits engineering hours that have been put into an Ethereum-Geth itself — makes it a system that's far more accessible for auditors and contributors, giving us strong confidence in its implementation security that will only harden as the ecosystem grows. Edit this page Last updatedonApr 29, 2024Previous Troubleshooting: Arbitrum bridge Next Overview: The Lifecycle of an Arbitrum Transaction