Architecture overview

There are four main steps for bringing a Stylus program to life:coding, activation, execution, and proving .

Coding

Developers can now write smart contracts in any programming language that compiles to WASM.

note Some high-level languages generate far more performant WASMs than others. Currently, there is support for Rust, C, and C++. However, the levels of support vary. Rust has rich language support from day one, with an open-source SDK that makes writing smart contracts in Rust as easy as possible. C and C++ are supported off the bat, enabling the deployment of existing contracts in those languages on-chain with minimal modifications.

The Stylus SDK for Rust contains the smart contract development framework and language features most developers will need to use in Stylus. The SDK also makes it possible to perform all EVM-specific functionalities that smart contract developers use. Check out theRust SDK Guide and theCrate Docs .

Activation

Starting from a high-level language (such as Rust, C, or C++), the first compilation stage happens using the CLI provided in the Stylus SDK for Rust or any other compiler, such as Clang for C and C++. Once compiled, the WASM is posted onchain. Then, in an activation process, WASM gets lowered to a node's native machine code (such as ARM or x86).

Activating a Stylus program requires a new precompile, ArbWasm. This precompile produces efficient binary code tailored to a node's native assembly. During this step, a series of middlewares ensure that user programs execute safely and are deterministically fraud-proven. Instrumentation includes gas metering, depth-checking, memory charging, and more to guarantee all WASM programs are safe for the chain to execute. Stylus contracts can be called only after activation.

Gas metering is essential for certifying that computational resources are paid for. In Stylus, the unit for measuring cost is calledink , which is similar to Ethereum's gas but thousands of times smaller. There are two reasons why a new measurement is used: First, WASM execution is so much faster than the EVM that executing thousands of WASM opcodes could be done in the same amount of time it takes the EVM to execute one. Second, the conversion rate of ink to gas can change based on future hardware or VM improvements. For a conceptual introduction to Stylus gas and ink, seegas and ink (Stylus) .

Execution

Stylus programs execute in a fork ofWasmer , the leading WebAssembly runtime, with minimal changes to optimize their codebase for blockchain-specific use cases. Wasmer executes native code much faster thanGeth executes EVM bytecode, contributing to the significant gas savings that Stylus provides.

EVM contracts continue to execute the same way they were before Stylus. When calling a contract, the difference between an EVM contract and a WASM program is visible via anEOF -inspired contract header. From there, the contract executes using its corresponding runtime. Contracts written in Solidity and WASM languages can make cross-contract calls to each other, meaning a developer never has to consider which language the contract is in. Everything is interoperable.

Proving

Nitro operates in two modes: a "happy case" where it compiles execution history to native code, and a "sad case" during validator disputes, where it compiles execution history to WASM for interactive fraud proofs on Ethereum. Stylus builds on Nitro's fraud-proving technology, allowing it to verify both execution history and WASM programs deployed by developers.

Stylus is made possible by Nitro’s ability to replay and verify disputes using WASM. Validators bisect disputes until an invalid step is identified and proven on-chain through a“one-step proof.” . This deterministic fraud-proving capability ensures the correctness of any arbitrary program compiled to WASM. The combination of WASM's and Nitro's properties enables this technological leap we call Stylus.

For more details on Nitro’s architecture, refer to thedocumentation or theNitro whitepaper .

Why does this matter?

Stylus innovates on many levels, with the key ones described here:

One chain, many languages

There are roughly 20k Solidity developers, compared to 3 million Rust developers or 12 million C developers[1].

Developers can now use their preferred programming language, which is interoperable on any Arbitrum chain with Stylus. By onboarding the next million developers, scaling to the next billion users becomes possible.

A better EVM

Stylus' MultiVM brings the best of both worlds. Developers still get all of the benefits of the EVM, including the ecosystem and liquidity, while getting efficiency improvements and access to existing libraries in Rust, C, and C++, all without changing anything about how the EVM works. EVM equivalence is no longer the ceiling; it's the floor.

Cheaper execution

Stylus is a more efficient execution environment than the EVM, leading directly to gas savings for complex smart contracts. Computation and memory can be significantly cheaper. Deploying cryptography libraries can be done permissionlessly as custom application layer precompiles. Use cases that are impractical in the EVM are now possible in Stylus.

Opt-in reentrancy

Stylus doesn't just improve on cost and speed. WASM programs are also safer. Reentrancy is a common vulnerability developers can only attempt to mitigate in Solidity. Stylus provides cheap reentrancy detection, and using the Rust SDK, reentrancy is disabled by default unless intentionally overridden.

Fully interoperable

Solidity programs and WASM programs are completely composable. If working in Solidity, a developer can call a Rust program or rely on another dependency in a different language. If working in Rust, all Solidity functionalities are accessible out of the box. Edit this page Last updatedonJan 27, 2025 Previous Optimize WASM binaries Next Gas metering