

Platformatic

View online

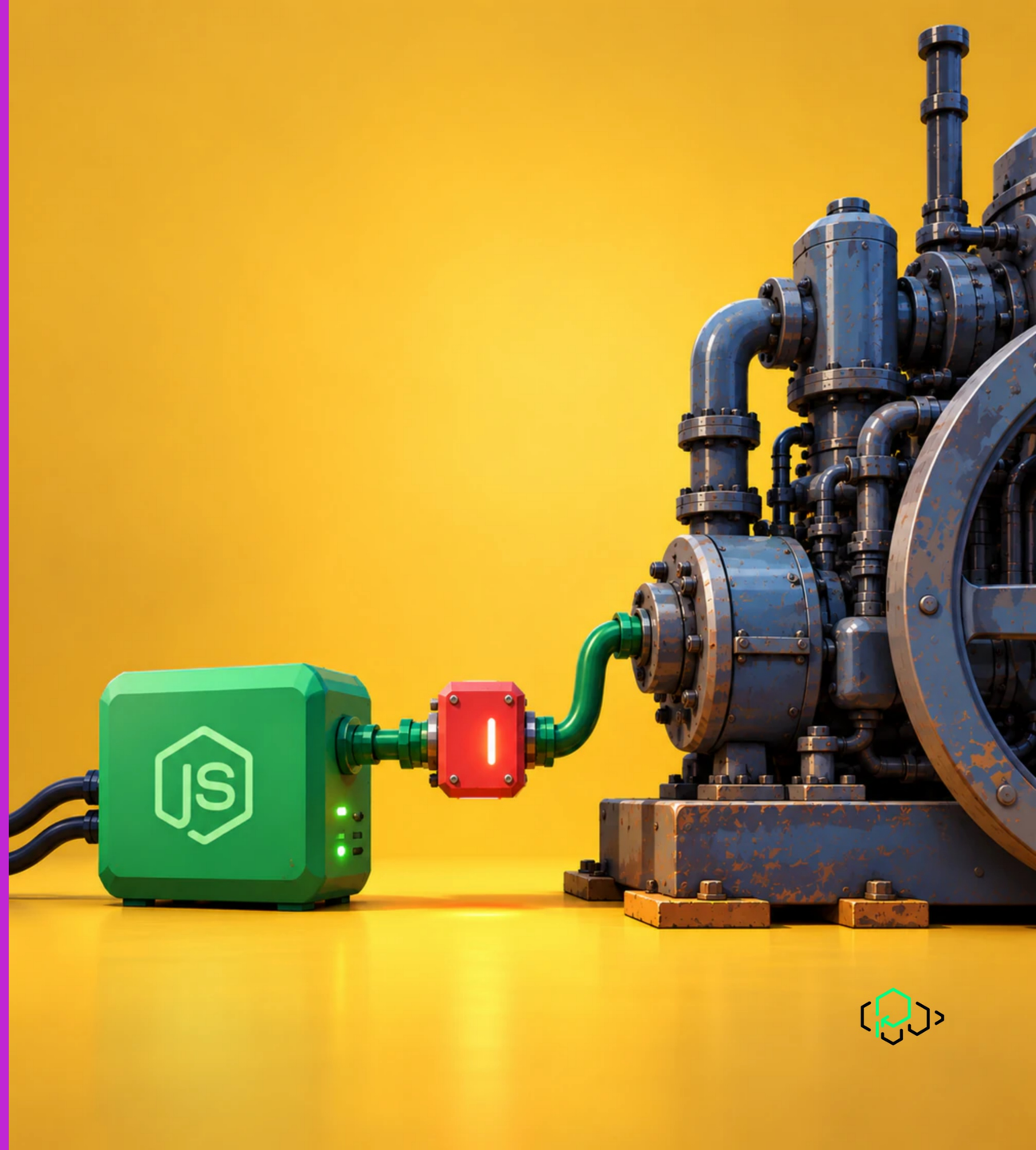
Download PDF

# FFI: Crossing the Native Boundary in Node.js

**Paolo Insogna**

Node.js TSC, Principal Engineer

# Not everything needs a rewrite.



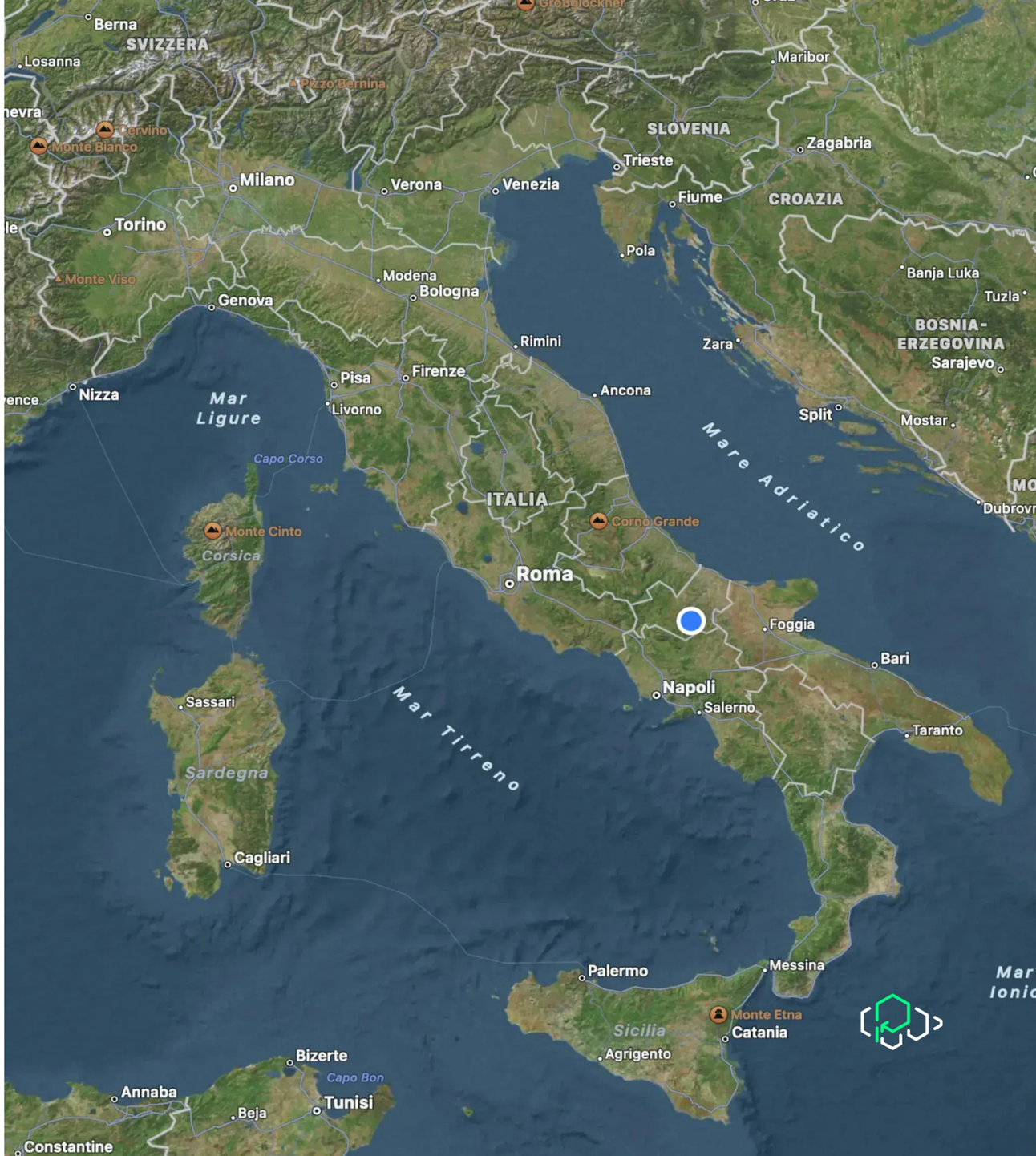
# Hello, I'm **Paolo**!



**Node.js** Technical Steering Committee Member

**Platformatic** Principal Engineer

`paoloinsogna.dev` `ShogunPanda` `p_insogna` `pinsogna`



# Some companies trusting us

Booking.com

Article

supabase

Case Study

SPENDESK

Case Study

# Your next dependency might not be in JavaScript.



# What is FFI?



**Foreign Function Interface**

Call functions written in one language from another.



**Same process, native code**

Load a shared library and invoke its exported symbols.



**A common binary contract**

C-compatible entry points can connect C, C++, Rust, Zig, and more.



# The ABI is the contract



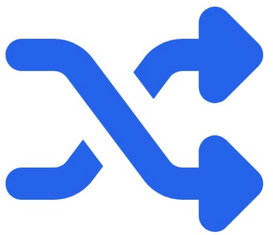
**Names and types**

Which symbol? Which arguments? Which return value?



**Binary representation**

Sizes, alignment, and memory layout must agree.



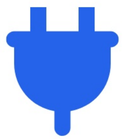
**Calling convention**

Each value must go in the correct register or stack slot.

# Why should we care?



**Reuse what already works**
Decades of native libraries do not need a JavaScript rewrite.



**Reach native capabilities**
Integrate device SDKs, codecs, and specialized libraries.



**Keep JavaScript in charge**
Coordinate the application while native code does its part.

# What can we build?

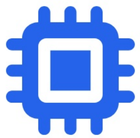


# What can we build?



**Media pipelines**

Use an existing image processor or audio engine.



**Hardware integrations**

Call a vendor's C API from a Node.js application.



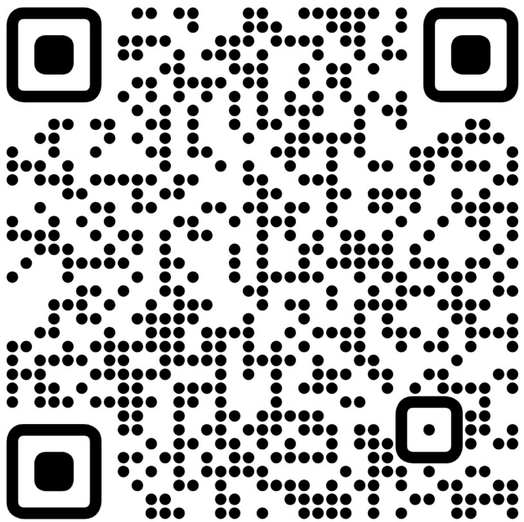
**And much more...**

Your imagination is the limit.



# Yes, even Doom.

Node.js, a native game engine, and a terminal. Because why not?

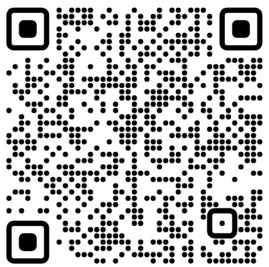


**https://github.com/platformatic/destino**

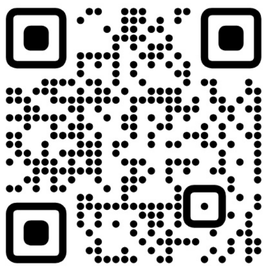
# Couldn't we just npm install it?



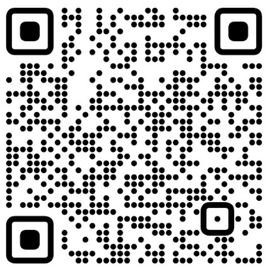
# Userland got here first



**node-ffi-napi**
Familiar API, with documented GC and multithreading caveats.



**Koffi**
Low-overhead calls, structs by value, and async support.



**node-ffi-rs**
Rust + Node-API, with its own memory and threading model.



# Why put FFI in core?



**One fewer native addon to distribute**

The FFI bridge ships with Node.js. Your target library still needs a compatible binary.



**Runtime integration**

Coordinate V8 fast calls, environments, permissions, and releases in one project.



**Portability maintained in core**

Test the FFI bridge and its ABI support alongside Node.js on supported platforms.

# Let's get to the action!



# Start with a native function

```c
#include <stdint.h>

// Export this symbol from the shared library.
int32_t add_i32(int32_t a, int32_t b) {
  return a + b;
}
```



# Load. Describe. Call.

```
import { dlopen, suffix } from 'node:ffi'

using handle = dlopen(`./mylib.${suffix}`, {
  add_i32: {
    arguments: ['int32', 'int32'],
    return: 'int32'
  }
})

// SAFETY: The signature matches the exported C function.
console.log(handle.functions.add_i32(20, 22)) // 42
```






# A small API. A long journey.

# 2023: Bryan starts crossing



**Bryan English (@bengl)**
Original implementation: **PR #46905**.



**Experience from sbffi**
From a userland experiment using dyncall to a core proposal using libffi.



**Portability shapes the design**
The backend has to work across the platforms Node.js supports.

# The hard part was never just calling C



**Who owns the memory?**
A pointer gives you an address, not a lifetime or a size.

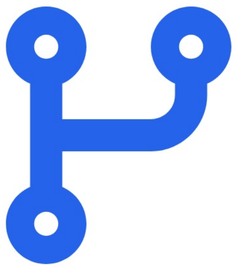


**Can native code call us back?**
JavaScript callbacks add thread, lifetime, and reentrancy constraints.



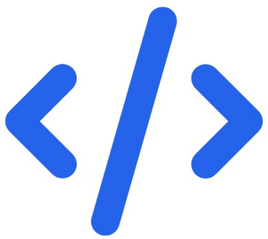
**Does it build everywhere?**
ABI differences and native dependencies become runtime maintenance work.

# 2025: the idea keeps moving



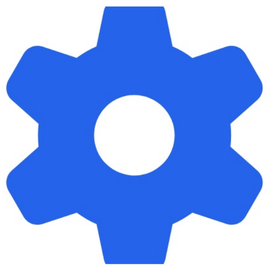
**New iteration by tianxiadys**

**PR #57761** explicitly built on Bryan's work.



**A different API exploration**

UnsafePointer, UnsafePointerView, and UnsafeCallback.



**More integration questions**

Callbacks, permissions, and bundled libffi.

# 2026: Colin restarts the conversation



**Colin Ihrig (@cjihrig)**

Opened **PR #62072** on March 2nd, 2026, renewing the initiative.



**From exploration to review**

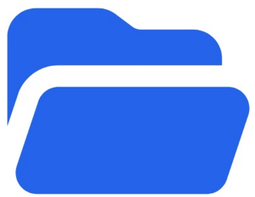
The API and implementation evolved through collaboration and platform testing.



**Landed on April 14th, 2026**

**node:ffi** became part of Node.js core.

# Finding the API that fits Node.js



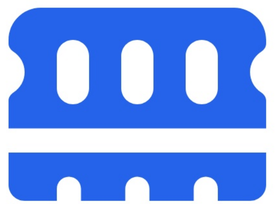
**DynamicLibrary**

Keep symbols, callable wrappers, and callbacks attached to a library.



**Explicit signatures**

Describe arguments and return types with simple objects.



**Explicit memory operations**

Copy bytes, borrow views, or access raw pointers.



# Welcome, `node:ffi`.

**Introduced as experimental in Node.js 26.1.0.**

# How do I use it?



# Types are part of the call

```
// C: uint64_t identity_u64(uint64_t value);
const identity = lib.getFunction('identity_u64', {
  arguments: ['uint64'],
  return: 'uint64'
})

// SAFETY: The declaration matches the exported symbol.
const value = identity(9_007_199_254_740_993n)
console.log(value) // 9007199254740993n

// A number would already have lost precision.
```



# With great power comes great responsibility™



# A pointer is not an ownership model



**JavaScript to native**

Buffers can lend their backing memory for the call; strings use temporary UTF-8 storage.



**Native to JavaScript**

toBuffer copies by default; a zero-copy view still depends on native memory.



**Lifetime is your contract**

Keep memory alive and stable for as long as either side can access it.

# Borrow bytes, not ownership

```c
#include <stdint.h>

uint32_t sum_bytes(const uint8_t *bytes, uint32_t length) {
  uint32_t total = 0;
  for (uint32_t i = 0; i < length; i++) {
    // SAFETY: The caller provides length readable bytes.
    total += bytes[i];
  }
  return total;
}
```



# A Buffer crosses the boundary

```
import { dlopen, suffix } from 'node:ffi'

using handle = dlopen(`./mylib.${suffix}`, {
  sum_bytes: {
    arguments: ['buffer', 'uint32'],
    return: 'uint32'
  }
})
const bytes = Buffer.from([10, 20, 12])

// SAFETY: Live storage; C reads bytes.length bytes, retaining none.
const total = handle.functions.sum_bytes(bytes, bytes.length)
console.log(total) // 42
```



# Cleanup has an order

1 **Stop native activity**
Make sure native code will no longer invoke callbacks or use borrowed memory.

2 **Release owned resources**
Use the library's matching release functions while its code is still loaded.

3 **Close the library**
DynamicLibrary supports `close()` and `Symbol.dispose`. Raw pointers are not revoked.



# Callbacks cross the other way



`registerCallback(signature, callback)`
Create a native function pointer backed by JavaScript through a libffi closure.



**Same thread, synchronous result**
Invoke on the creating thread. Do not throw or return a Promise.



**Unregister after the last native use**
Never unregister an active callback or close its library while it is running.

# C calls JavaScript, then returns

```c
#include <stdint.h>

typedef int32_t (*transform_fn)(int32_t);

int32_t apply_once(int32_t value, transform_fn transform) {
  // SAFETY: A live callback is invoked on its creating thread.
  return transform(value);
}
```



# Register. Call. Unregister.

```
import { dlopen, suffix } from 'node:ffi'

using handle = dlopen(`./mylib.${suffix}`, {
  apply_once: {
    arguments: ['int32', 'function'],
    return: 'int32'
  }
})
const callback = handle.lib.registerCallback(
  { arguments: ['int32'], return: 'int32' }, value => value
)

try {
  // SAFETY: Synchronous, same-thread; no throw or Promise.
  console.log(handle.functions.apply_once(42, callback)) // 42
} finally {
  // SAFETY: C has returned without retaining the callback.
  handle.lib.unregisterCallback(callback)
}
```



# Native does not mean asynchronous



**A synchronous call blocks its thread**

A slow native function can still stall the main event loop.



**Move suitable work to a worker**

Load and manage the library there, respecting its own thread-safety rules.



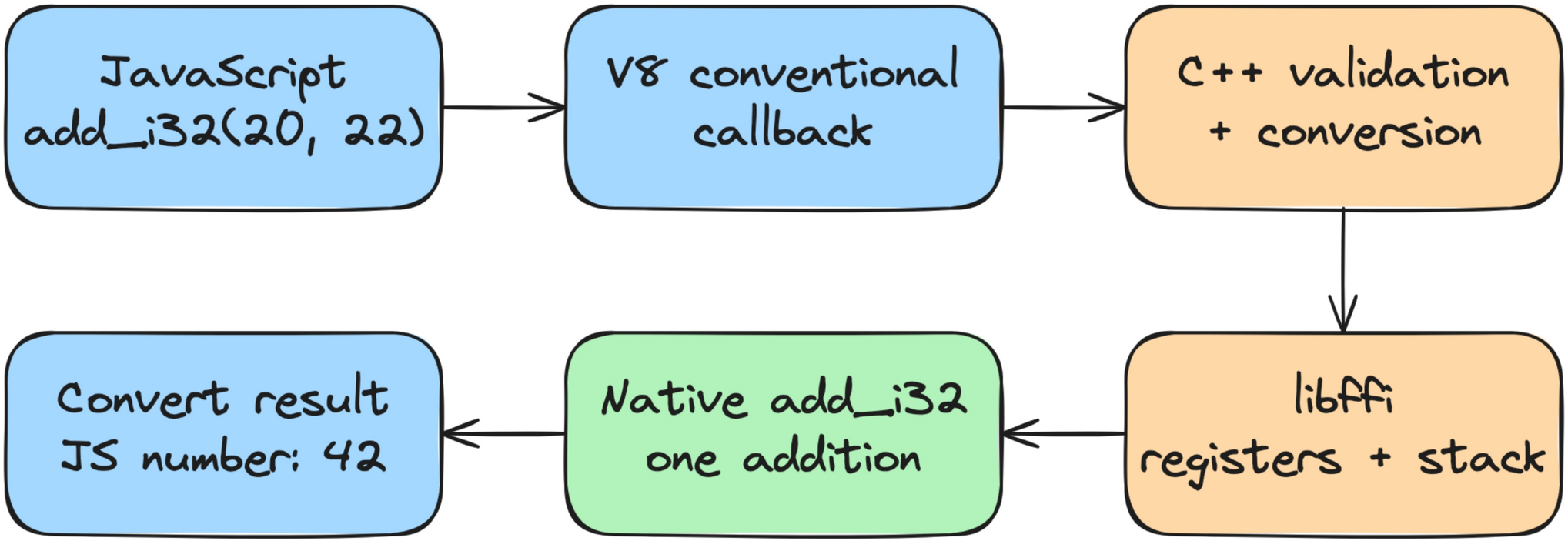
`getCurrentEventLoop()`

Exposes the current uv_loop_t for native integration, not automatic offloading.

# It works.
# But how fast?



# Every crossing has a cost



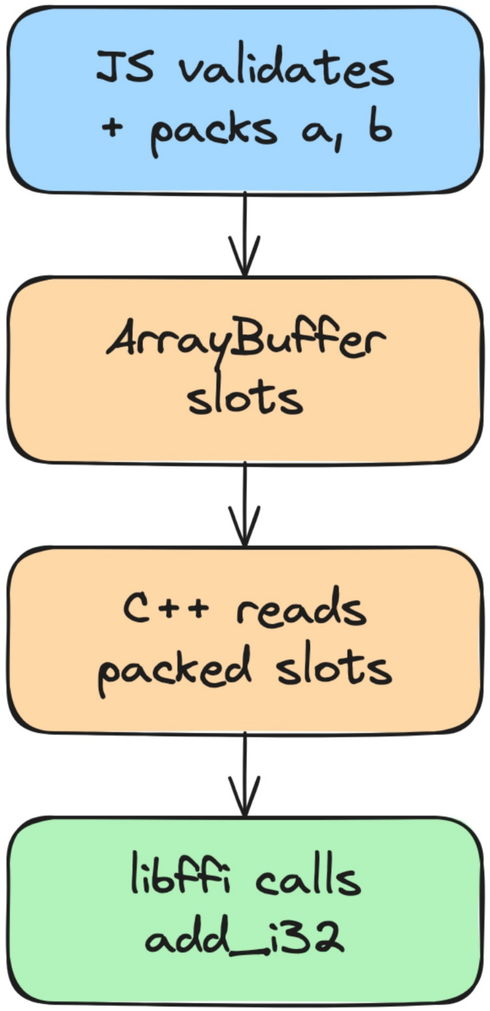

For a tiny native function, the crossing can dominate the work.

# First optimizations

**PR #62918** by **Bryan English**

Less overhead, Still using libffi.

One ArrayBuffer per function,
reused by JS and C++.



# Pack once per call, reuse the storage

```
// Simplified internals: one reusable buffer per function.
const slots = new DataView(new ArrayBuffer(3 * 8))

// Each call validates and packs the arguments.
slots.setInt32(8, 20, true)
slots.setInt32(16, 22, true)

// C++ snapshots the slots; libffi calls add_i32.
invokeNativeWithoutJSArguments()

const result = slots.getInt32(0, true) // 42
```

# V8 already knows a faster route

Fast API can call a typed native entry point directly.

But this C function does not expect V8's receiver.



# One unexpected argument.

The JavaScript receiver.

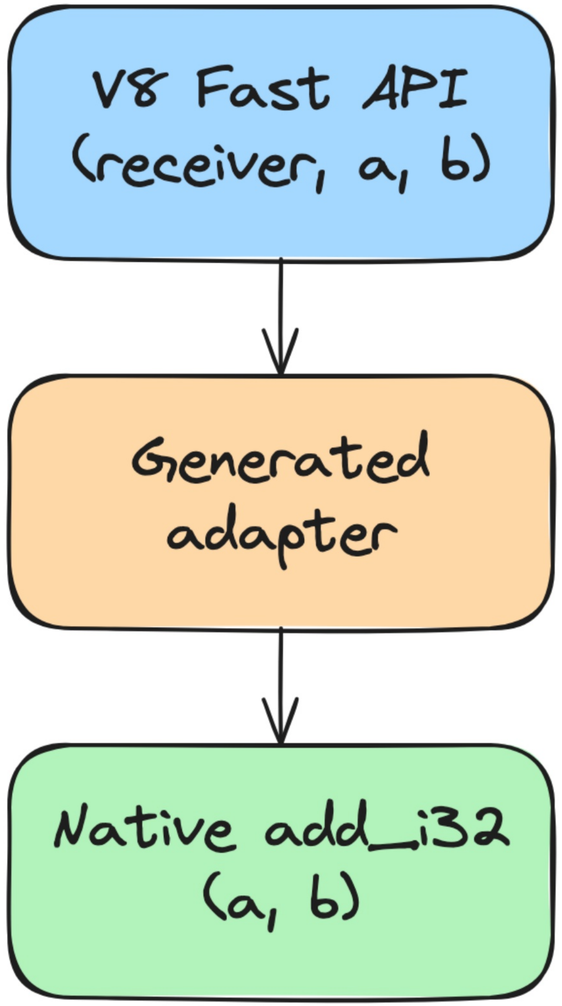


# A tiny adapter bridges the gap

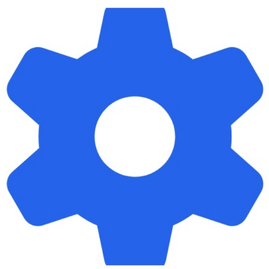
The adapter repositions a and b, omitting the receiver. The result is still 42.

Prepared once: allocate RW → emit code → flush instruction cache → protect RX.

RW = read/write; RX = read/execute.
Fast calls reuse the RX code.

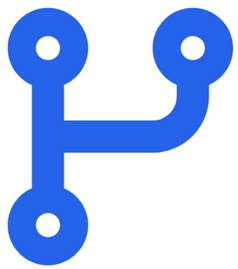


# We explored more than one route



**Generate wrappers with a compiler**

Cranelift, MIR, and more.



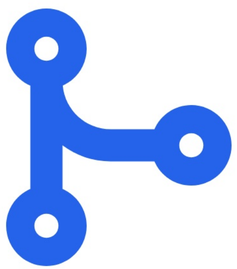
**Patch V8**

A clever suggestion from Anna Henningsen.

# So, what happened?



# We brought it home!



**PR #63068 — June 16th, 2026**

My Fast FFI implementation, co-authored with Bryan, landed in Node.js.



**One public API**

Eligible calls can use Fast API.
No visible changes visible outside.

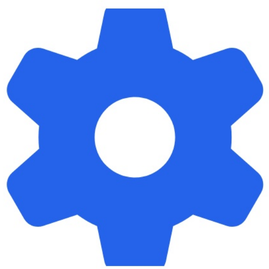
# The adapter, in pseudocode

```
; ARM64 pseudocode; uppercase operations are symbolic.
; x0 = receiver, w1 = a, w2 = b. wN = low 32 bits of xN.
; sp = stack pointer; x29 = frame pointer; x30 = return address.

stp x29, x30, [sp, #-16]! ; sp -= 16; save x29 and x30.
CHECK_LIBRARY_OPEN        ; If closed: JS error, restore, return.
mov x0, x1                ; Move a over the receiver.
mov x1, x2                ; Move b into C's second argument.
LOAD_ADDRESS x16, add_i32 ; Load the resolved target address.
; SAFETY: Open library; matching native signature.
blr x16                   ; Call target; set x30; result in w0.
ldp x29, x30, [sp], #16   ; Restore x29/x30; sp += 16.
ret                       ; Return to V8 via x30; keep w0.
```



# Prepared once. Reused on every fast call.



### Prepare the adapter

When creating an eligible function, Node.js generates an adapter for the platform.



### Reuse it

Optimized calls reuse that adapter. Code generation does not happen on every call.

# Three paths, one contract

1 **Generic**
Convert arguments in C++ and call through libffi.

2 **Shared buffer**
Prepare arguments in JavaScript and still call through libffi.

3 **Fast API**
Optimized JavaScript calls through a generated adapter.

# Fallback is part of the design.




Was it worth it?

# Reading the benchmark columns



**ffi-rs / Koffi**

Third-party FFI libraries for Node.js.



**Generic / Shared**

C++ argument conversion vs a reusable JS/C++ argument buffer; both call through libffi.



**Fast API**

V8 fast calls through a generated native adapter, when eligible.



# How fast is Fast FFI?

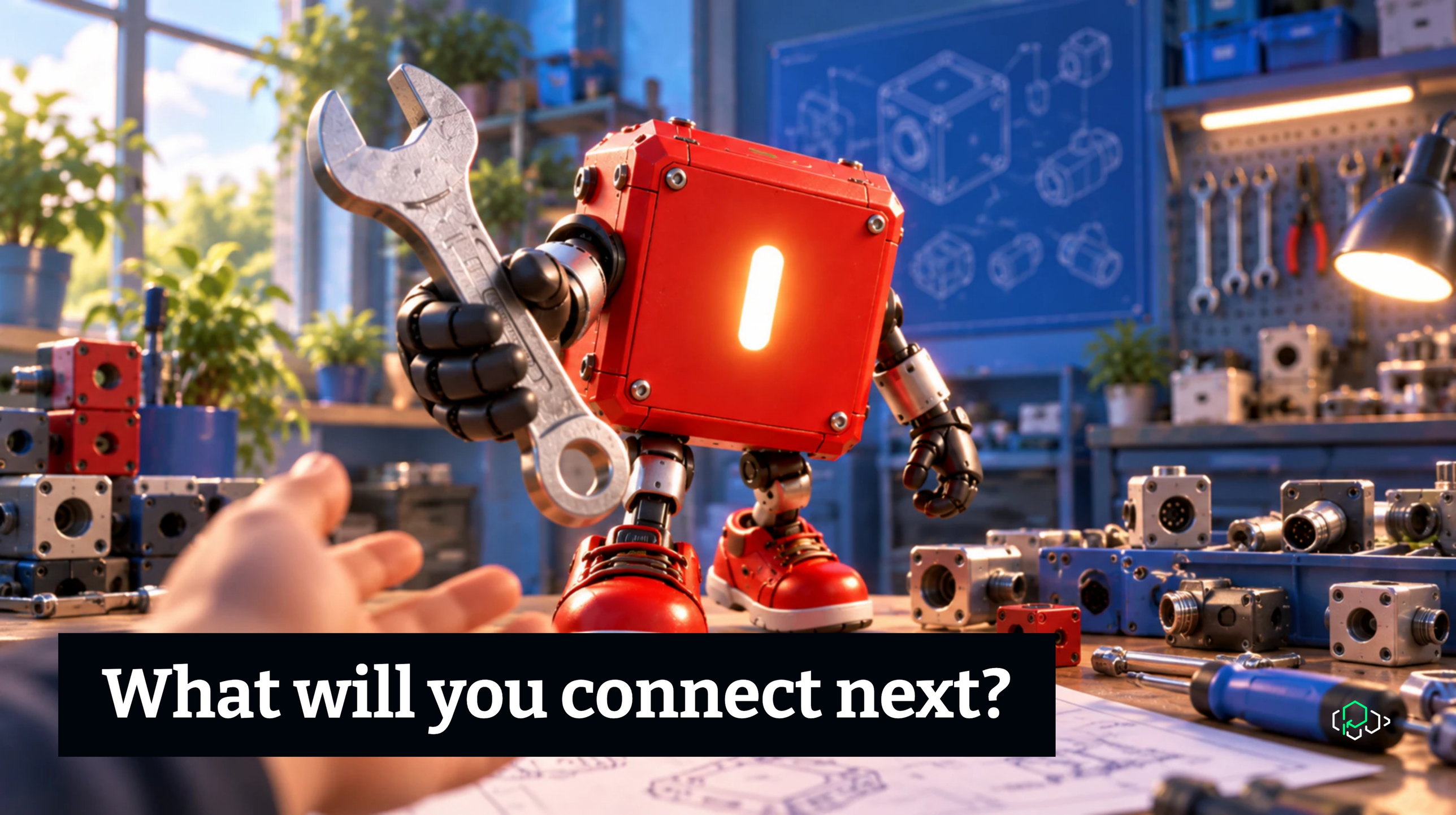
Benchmarks run on a Apple M2 Max · Median ns/call, lower is better · 15 runs

| Benchmark | ffi-rs | Koffi | Generic | Shared | Fast API |
|---|---|---|---|---|---|
| identity-i32 | 1638.61 | 22.08 | 78.12 | 53.76 | 2.13 |
| add-i32 | 2020.85 | 26.65 | 87.73 | 70.25 | 2.21 |
| sum-8-i32 | 4178.95 | 51.51 | 122.69 | 124.65 | 124.73 |
| sum-buffer: 64 B | 2060.27 | 43.42 | 107.78 | 83.78 | 14.04 |
| sum-buffer: 1 KiB | 2094.26 | 97.19 | 161.94 | 138.32 | 68.80 |
| sum-buffer: 16 KiB | 2986.33 | 964.40 | 1037.58 | 1006.89 | 931.89 |

# Let's focus on Node.js Fast API vs Koffi

Node.js wins on the fast-path cases measured here; Koffi wins on the eight-argument fallback.

| Benchmark | Throughput vs Koffi |
|---|---|
| identity-i32 | +937.09% |
| add-i32 | +1105.01% |
| sum-8-i32 | -58.70% |
| sum-buffer: 64 B | +209.20% |
| sum-buffer: 1 KiB | +41.25% |
| sum-buffer: 16 KiB | +3.49% |


What will you connect next?




# Challenge won!

# One last thing™

*“The best way to predict the future is to invent it.”*

**Alan Kay**





# Thank you!

**Paolo Insogna**

**Node.js TSC, Principal Engineer**

@p_insogna

paolo.insogna@platformatic.dev

Platformatic