View online

Download PDF

Milo, a new HTTP parser for Node.js

Paolo Insogna

Node.js TSC, Principal Engineer @ **Platformatic**

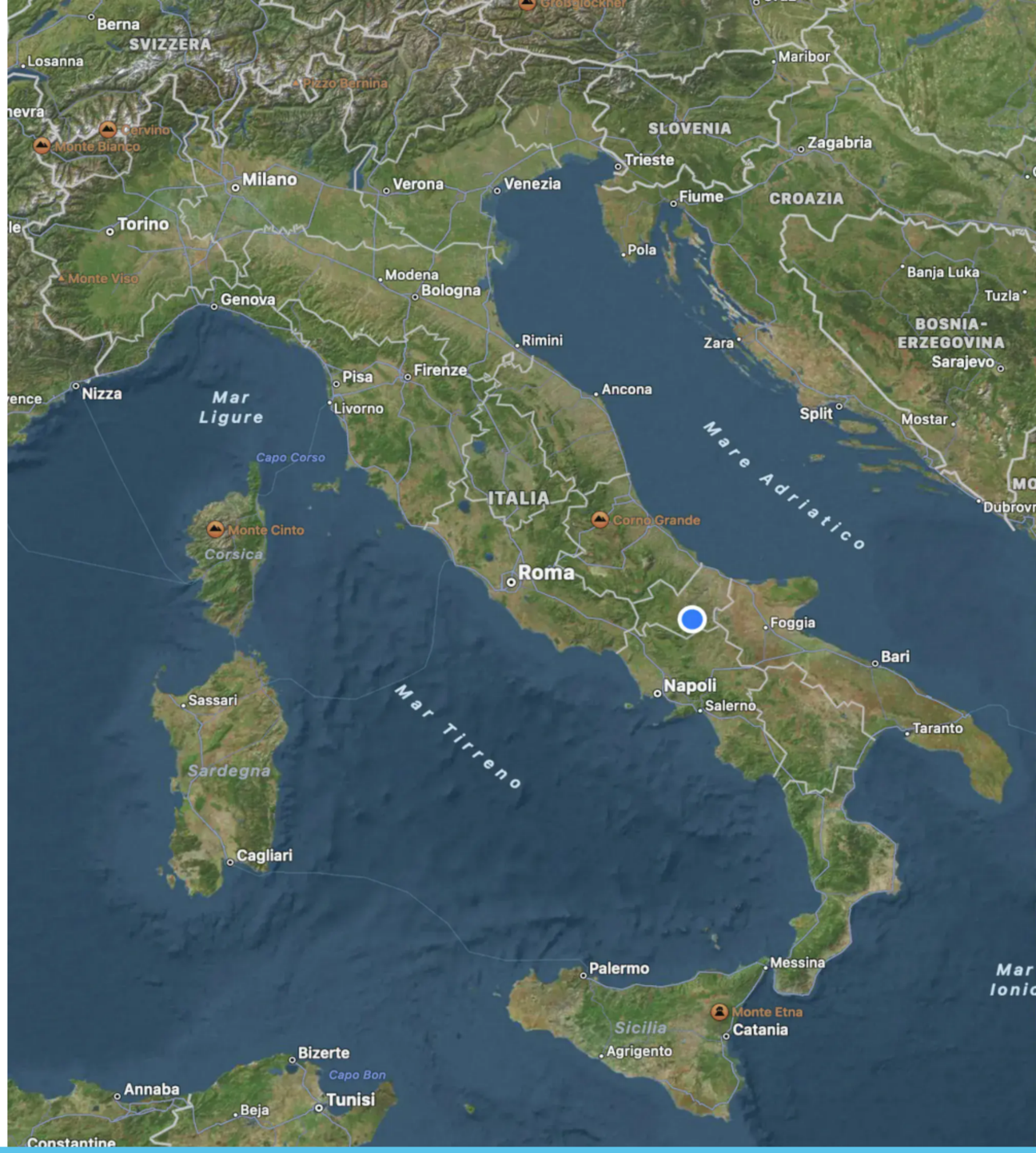
Being reckless (sometimes) pays off!

Hello, I'm **Paolo**!

Node.js Technical Steering Committee Member

Platformatic Principal Engineer

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



We all love HTTP!

Which HTTP are you?

The choice is narrow

Even if the HTTP protocol is more than 30 years old, only **three** current versions of it exist as of today. The others are considered obsolete.

1. **HTTP/1.1**
   The last version of the initial protocol. By far the most famous and most used.

2. **HTTP/2**
   Developed on top of SPDY to remove some problems of HTTP.

3. **HTTP/3**
   Developed on top of QUIC to solve TCP problems.

What about Node.js?

1+2

HTTP/1.1 and HTTP/2

Node.js has a stable implementation.

3

HTTP/3

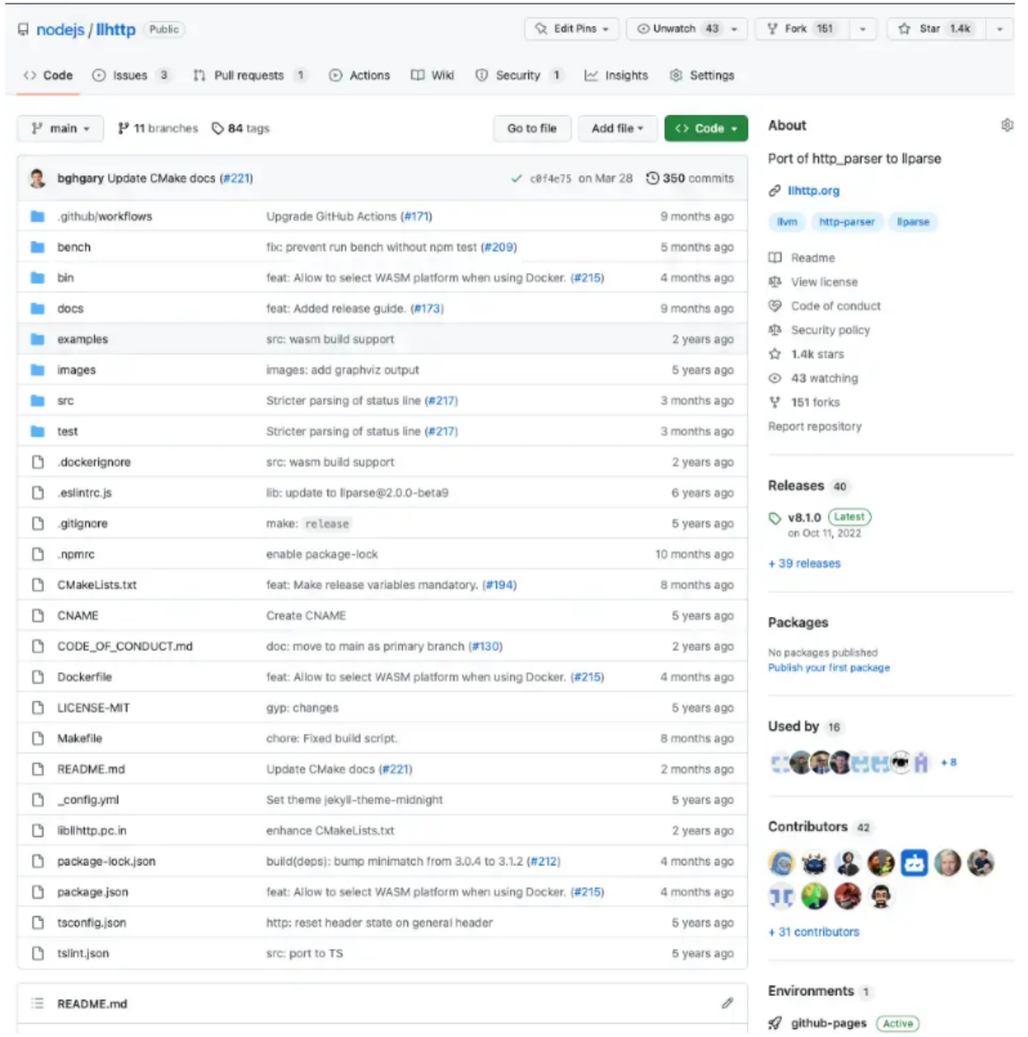
Work in progress, **stay tuned!**

Let's focus!

The current parser

llhttp is the current HTTP parser.

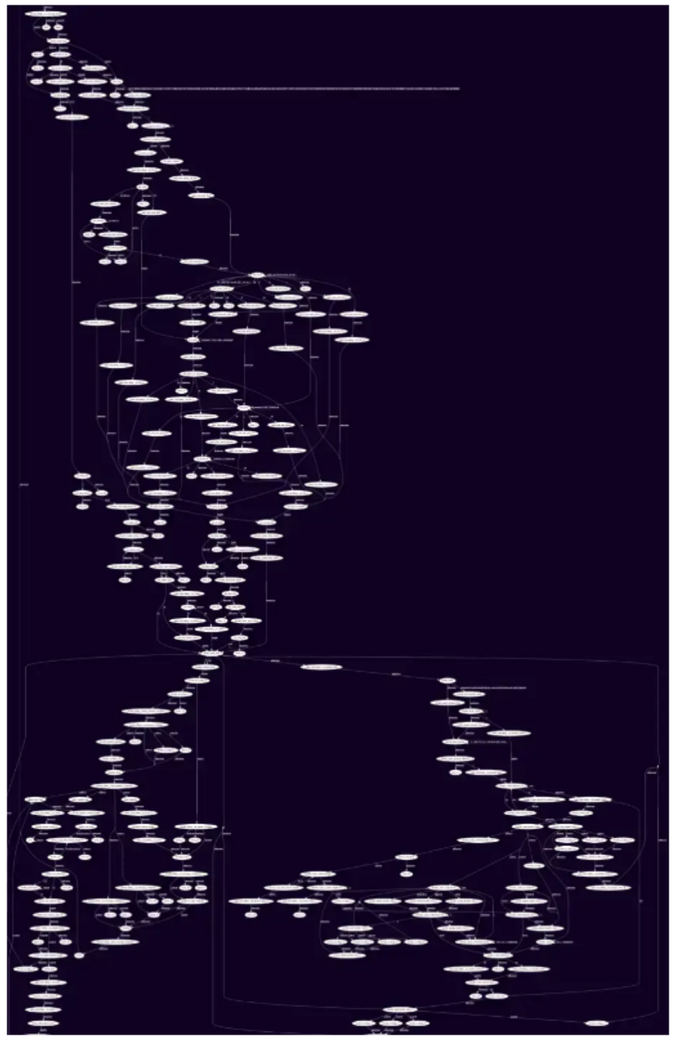
Written by Fedor Indutny in 2019, is the default since Node.js 12.



How it works?

llhttp is a state based HTTP parser based on llparse.

llparse is capable to generate a very performant C code out of a TypeScript description of the possible states.



llhttp: what's wrong with it?

Hard to debug and release

The transpilation makes hard to debug issues.

Backward compatibility

Supporting obsolete versions of HTTP introduces unneeded complexity.

You give them a finger, they take the arm

Leniency-prone approach opens the door for edge cases and vulnerabilities opportunities.

Do we have the solution?

Yes, start fresh!

Say hello to **Milo**!

Let's drop the bomb!

Milo is written in Rust

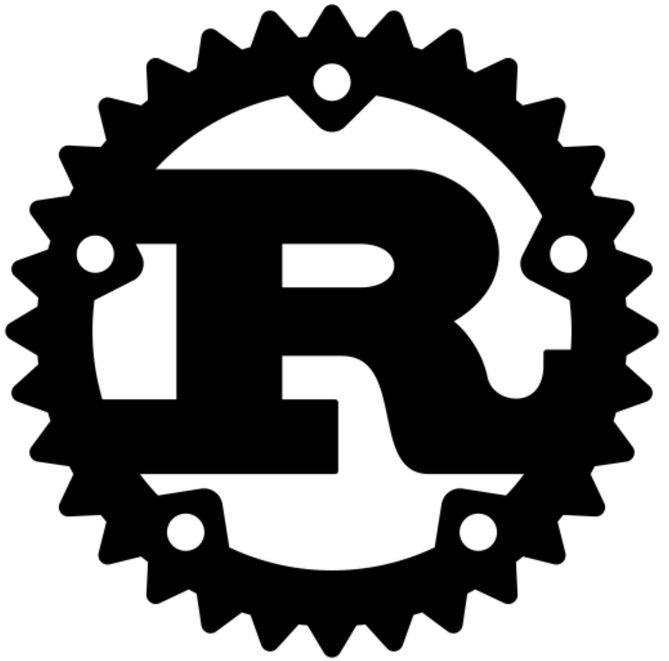
The language has proven flexible and performant to achieve the task.

I did not know it before Milo

I purposely made the choice to see how hard it is for a new prospect contributor.

Be cool

I don't want to start another language flame.
Please. ❤️



Do not throw the goods away

llhttp is a piece of art

It deeply inspires Milo and I kept most of his architecture. **Kudos to Fedor!**

Still a state machine, but simpler

llhttp has **80** possible states, Milo only **32**.

Declarative, reinvented

Rust enables to declare states with no code restriction.

How is that possible?

It's all in the macros!

Rust macro system is insanely powerful

It is evaluated at compile time

The executed code is inherently optimized.

No code limitation

As long as you return valid Rust code, everything is permitted.

Easily debuggable

Via **cargo-expand**, it's easy to see what is the final compiled code.

Examples are worth more than 1000 words

```
state!(request_protocol, {
  match data {
    string!("HTTP/") | string!("RTSP/") => {
      callback!(on_protocol, 4);
      parser.position += 4;

      move_to!(request_version, 1)
    }
    otherwise!(5) =>
      fail!(UNEXPECTED_CHARACTER, "Expected protocol"),
    _ => suspend!(),
  }
});
```

```
#[inline(always)]
pub fn state_request_protocol(
  parser: &mut Parser, data: &[c_uchar],
) -> usize {
    let mut data = data;
    match data {
        [72u8, 84u8, 84u8, 80u8, 47u8, ..] |
        [82u8, 84u8, 83u8, 80u8, 47u8, ..] => {
            #[cfg(not(target_family = "wasm"))]
            {
                (parser.callbacks.on_protocol)(
                  parser, parser.position, 4,
                );
            };
            parser.position += 4;
            parser.move_to(STATE_REQUEST_VERSION, 1)
        }
        [_u0, _u1, _u2, _u3, _u4, ..] => {
            parser.fail(
              ERROR_UNEXPECTED_CHARACTER,
              "Expected protocol",
            )
        }
        _ => SUSPEND,
    }
}
```

What about resources?

Milo has very small memory footprint

(Almost) No copy eager parsing

Data is analyzed in place without copying it.

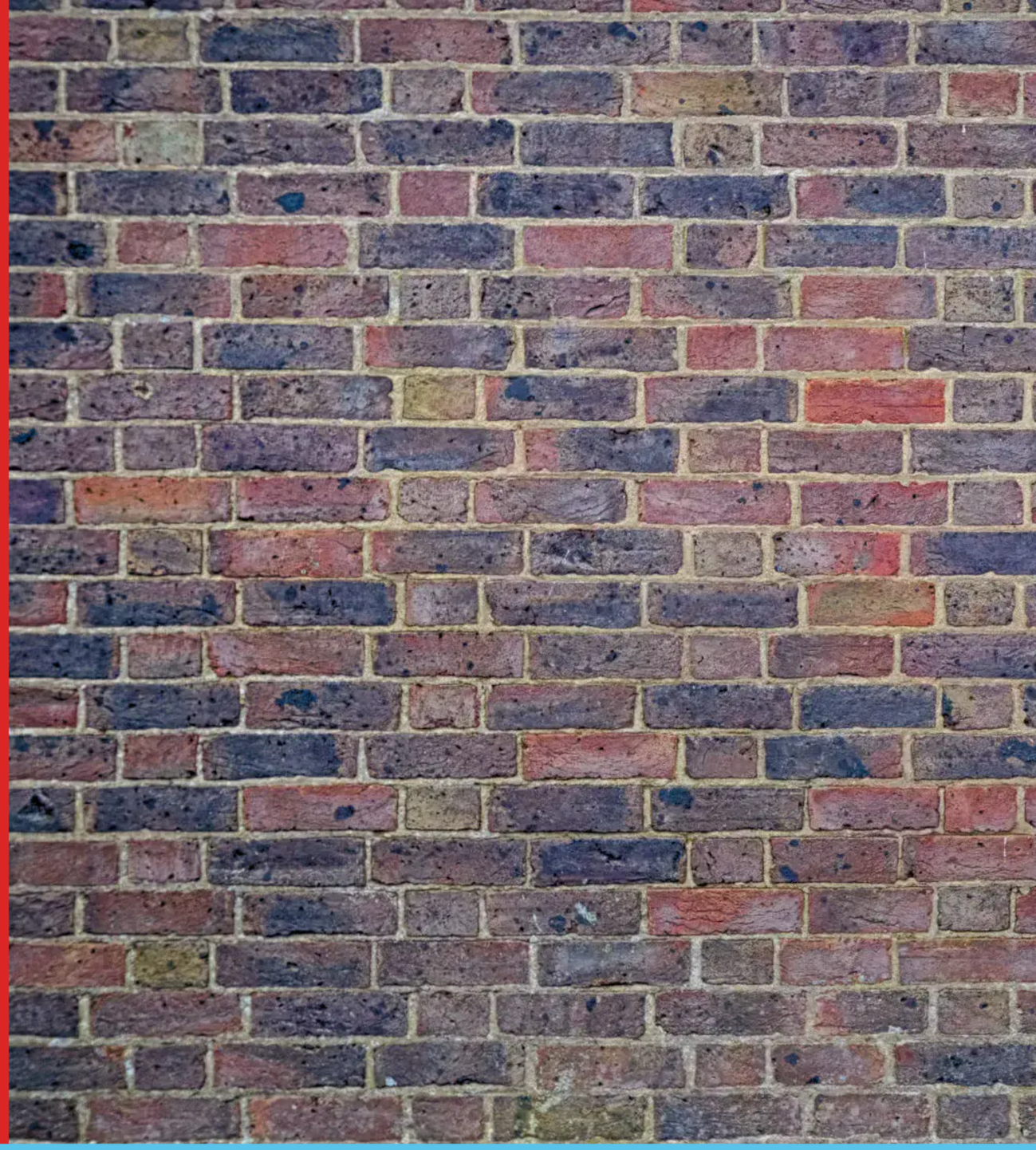
Only one exception (optional)

When parsing, only the unconsumed part of the input is copied in the parser.

Convenience at your service

Milo manages unconsumed data for you, making your life easier.

Strict, period!



Let's get to the action!

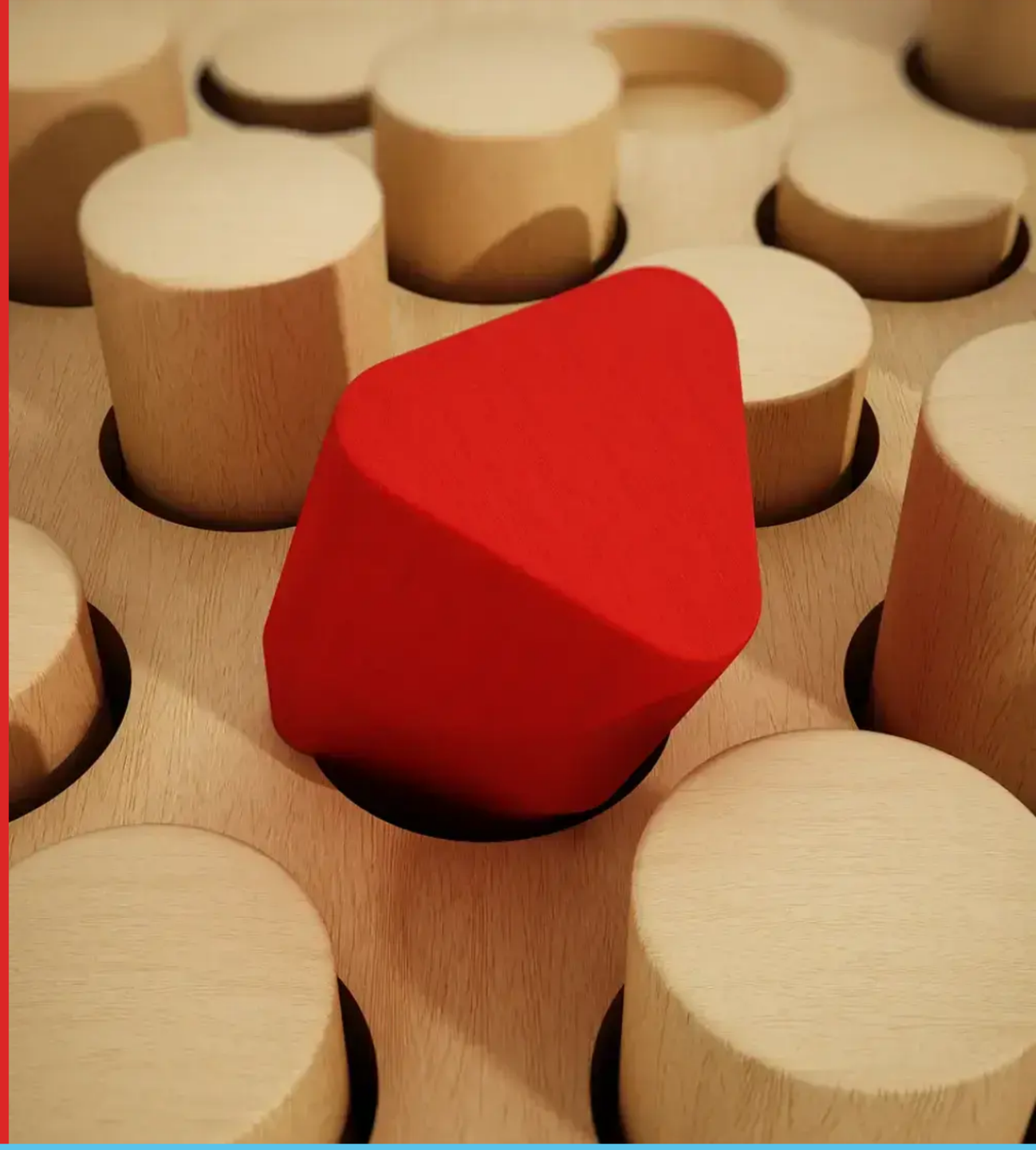
Sample code (Rust)

```rust
use core::ffi::c_void;
use core::slice;
use milo::Parser;

fn main() {
  // Create the parser.
  let mut parser = Parser::new();

  // Prepare a message to parse.
  let message = String::from("HTTP/1.1 200 OK\r\nContent-Length: 3\r\n\r\nabc");
  parser.context = message.as_ptr() as *mut c_void;

  // Milo works using callbacks, All callbacks have the same signature.
  parser.callbacks.on_data = |p: &mut Parser, from: usize, size: usize| {
    let message = unsafe {
      std::str::from_utf8_unchecked(slice::from_raw_parts(p.context.add(from) as *const u8, size))
    };

    // Do something with the informations.
    println!("Pos={} Body: {}", p.position, message);
  };

  // Now perform the main parsing using milo.parse.
  parser.parse(message.as_ptr(), message.len());
}
```

Output (Rust)

```
shogun@panda:~/example$ cargo run
Pos=38 Body: abc
```

But Node.js uses C++!



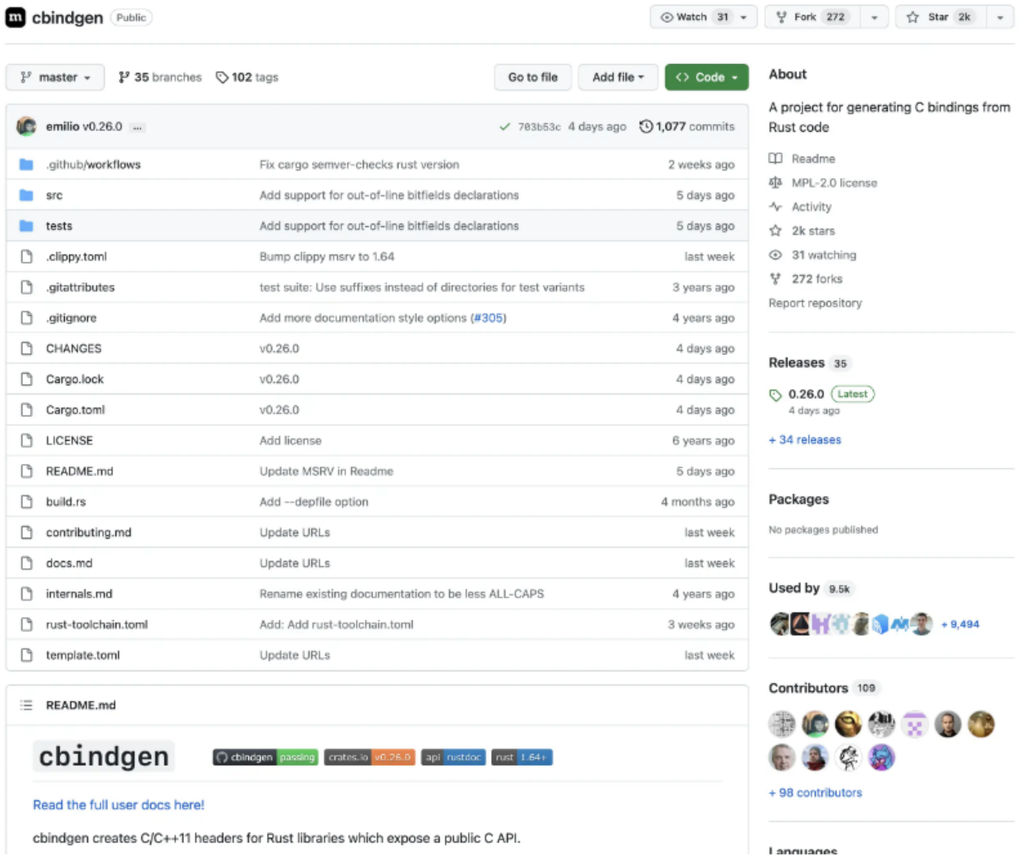
The C++ workflow

A tool generates the headers

cbindgen generates a fully working C or C++ header file. Only a small TOML file is needed.

Cargo generates a static library

The generated **libmilo.a** file can be statically linked in any C/C++ executable.



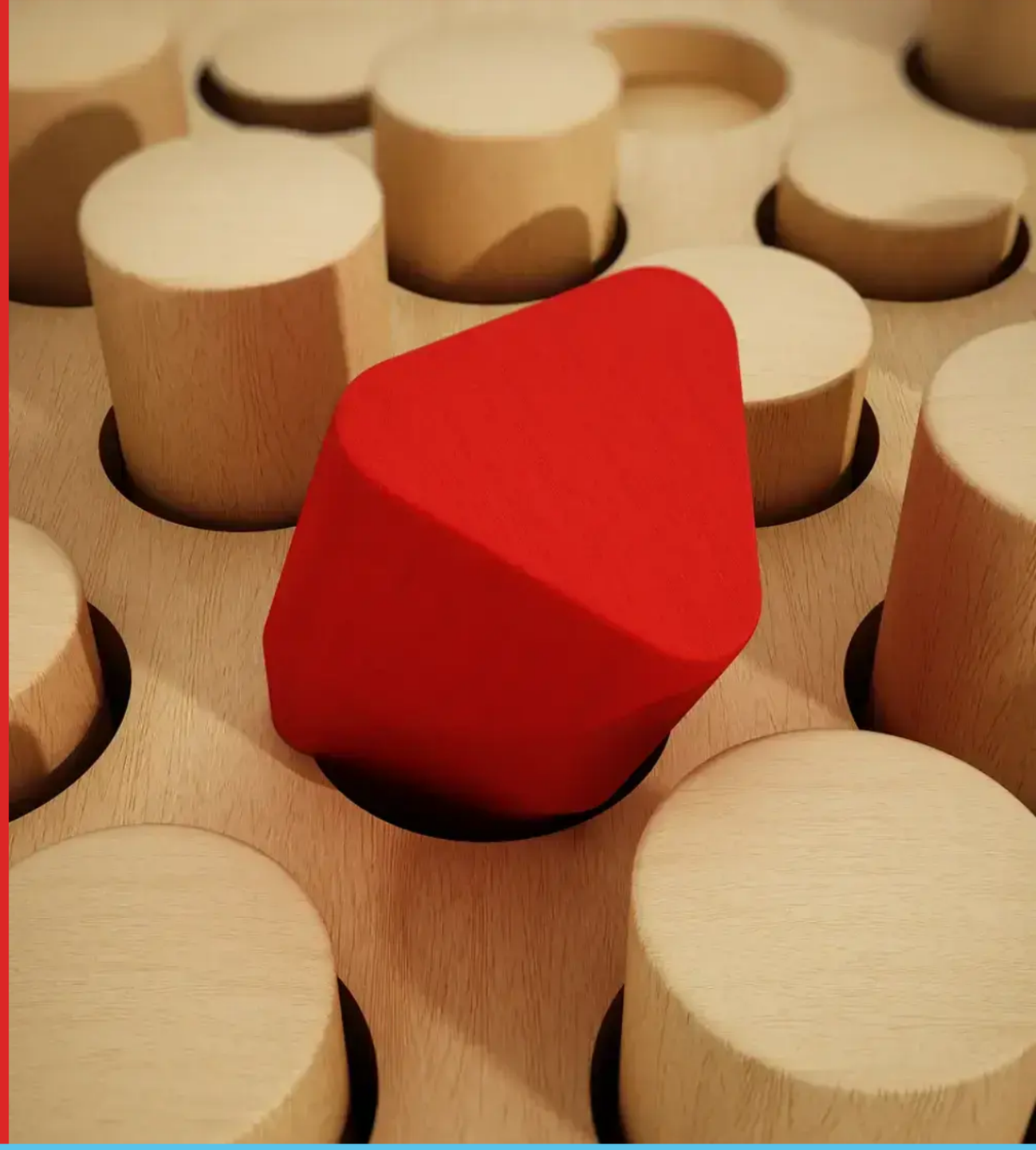
Sample code (C++)

```cpp
#include "milo.h"
#include "stdio.h"
#include "string.h"

int main() {
  // Create the parser.
  milo::Parser* parser = milo::milo_create();

  // Prepare a message to parse.
  const char* message = "HTTP/1.1 200 OK\r\nContent-Length: 3\r\n\r\nabc";
  parser->context = (char*) message;

  // Milo works using callbacks, All callbacks have the same signature.
  parser->callbacks.on_data = [](milo::Parser* p, uintptr_t from, uintptr_t size) {
    char* payload = reinterpret_cast<char*>(malloc(sizeof(char) * size));
    strncpy(payload, reinterpret_cast<const char*>(p->context) + from, size);
    printf("Pos=%lu Body: %s\n", p->position, payload);
    free(payload);
  };

  // Now perform the main parsing using milo.parse. The method returns the number of consumed characters.
  milo::milo_parse(parser, reinterpret_cast<const unsigned char*>(message), strlen(message));

  // Cleanup used resources.
  milo::milo_destroy(parser);
}
```

Output (C++)

```
shogun@panda:~/example$ clang++ -std=c++11 -o example libmilo.a main.cc
shogun@panda:~/example$ ./example
Pos=38 Body: abc
```

But I want to support SmartOS!



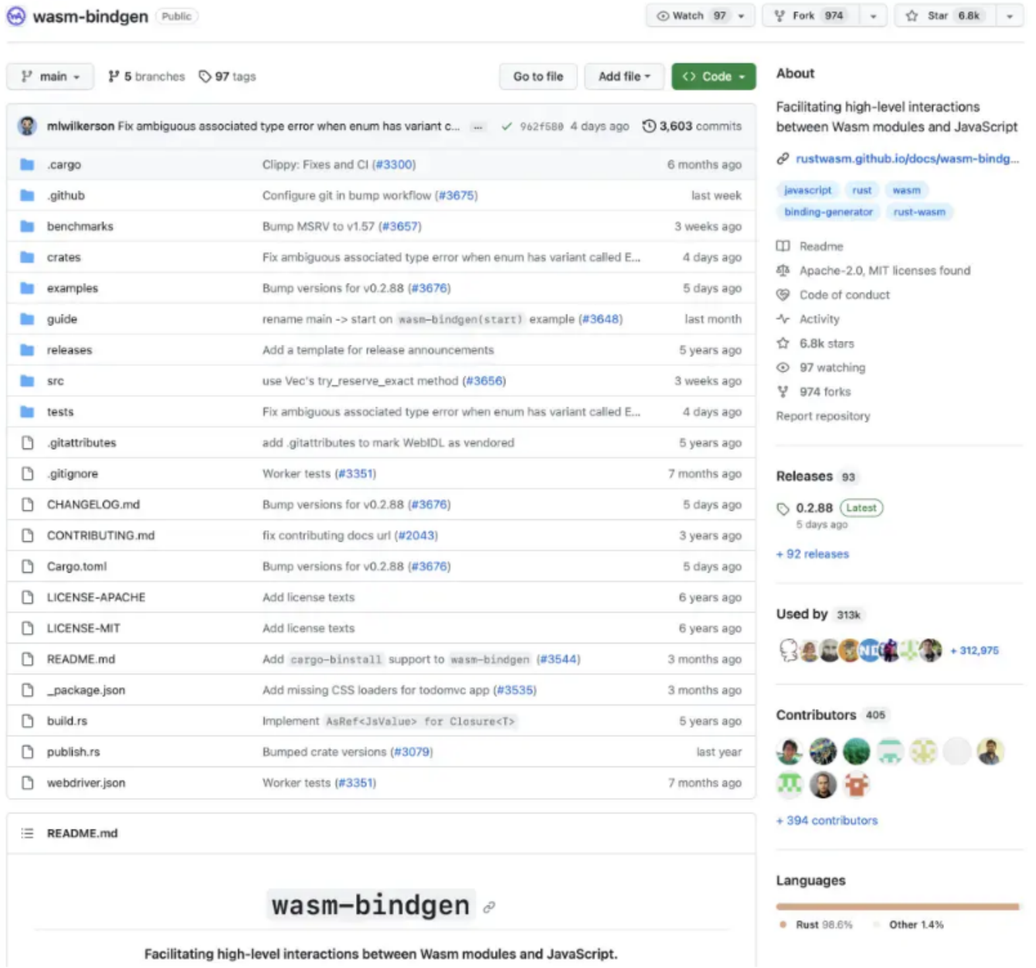
WASM will save us!

WebAssembly is fully supported

Rust has always considered WebAssembly a first class citizen.

The toolchain makes it easy

wasm-bindgen generates a fully working JS module which internally loads a WASM file.



Sample code (Node.js with WebAssembly)

```
import { milo } from '@perseveranza-pets/milo'

// Prepare a message to parse.
const message = Buffer.from('HTTP/1.1 200 OK\r\nContent-Length: 3\r\n\r\nabc')

// Allocate a memory in the WebAssembly space. This speeds up data copying to the WebAssembly layer.
const ptr = milo.alloc(message.length)

// Create a buffer we can use normally.
const buffer = Buffer.from(milo.memory.buffer, ptr, message.length)

// Create the parser.
const parser = milo.create()

// Milo works using callbacks, All callbacks have the same signature.
milo.setOnData(parser, (p, from, size) => {
  console.log(`Pos=${milo.getPosition(p)} Body: ${message.slice(from, from + size).toString()}`)
})

// Now perform the main parsing using milo.parse. The method returns the number of consumed characters.
buffer.set(Buffer.from(message), 0)
const consumed = milo.parse(parser, ptr, message.length)

// Cleanup used resources.
milo.destroy(parser)
milo.dealloc(ptr, message.length)
```

Output (Node.js with WebAssembly)

```
shogun@panda:~/example$ node index.mjs
Pos=38 Body: abc
```

And that's Milo!

Performance in Node (native, preliminary)

```
====================================== llhttp ======================================

-------------------------------------------------------------------
| Stat    | 2.5% | 50%  | 97.5% | 99%  | Avg     | Stdev   | Max  |
-------------------------------------------------------------------
| Latency | 0 ms | 0 ms | 0 ms  | 0 ms | 0.01 ms | 0.12 ms | 17 ms |
-------------------------------------------------------------------
-------------------------------------------------------------------------------
| Stat      | 1%      | 2.5%    | 50%     | 97.5%   | Avg     | Stdev   | Min     |
-------------------------------------------------------------------------------
| Req/Sec   | 25119   | 25119   | 29055   | 30447   | 28808   | 1411.49 | 25114   |
-------------------------------------------------------------------------------
| Bytes/Sec | 2.11 MB | 2.11 MB | 2.44 MB | 2.56 MB | 2.42 MB | 118 kB  | 2.11 MB |
-------------------------------------------------------------------------------

Req/Bytes counts sampled once per second.
# of samples: 11

317k requests in 11.02s, 26.6 MB read
```

```
======================================  milo  ======================================

-------------------------------------------------------------------
| Stat    | 2.5% | 50%  | 97.5% | 99%  | Avg     | Stdev   | Max  |
-------------------------------------------------------------------
| Latency | 0 ms | 0 ms | 0 ms  | 0 ms | 0.01 ms | 0.11 ms | 17 ms |
-------------------------------------------------------------------
-------------------------------------------------------------------------------
| Stat      | 1%      | 2.5%    | 50%     | 97.5%   | Avg     | Stdev   | Min     |
-------------------------------------------------------------------------------
| Req/Sec   | 28511   | 28511   | 30287   | 33407   | 30765.1 | 1458.2  | 28507   |
-------------------------------------------------------------------------------
| Bytes/Sec | 2.4 MB  | 2.4 MB  | 2.55 MB | 2.81 MB | 2.58 MB | 122 kB  | 2.39 MB |
-------------------------------------------------------------------------------

Req/Bytes counts sampled once per second.
# of samples: 11

338k requests in 11.02s, 28.4 MB read
```

What's missing?

Node.js integration

I just finished integrating the WebAssembly version in undici. Our beloved runtime is next.

SIMD in WebAssembly

Milo matches or outperforms llhttp in native mode, but it is slower when compiled in WebAssembly.

Migrate llhttp test suite

The llhttp test suite (originally from http_parser) is crucial to ensure correctness of the parser.

A due thanks to ...

Without their trust, support and sponsorship, Milo would have never been possible. **Love you!** ❤️

One last thing™

“A person who never made a mistake never tried anything new.”

Albert Einstein

Thank you!

Paolo Insogna

Node.js TSC, Principal Engineer

@p_insogna

paolo.insogna@platformatic.dev

Platformatic