Platformatic

View online

Download PDF

Node.js: More threads than you think

Paolo Insogna
Node.js TSC, Principal Engineer

There is a lot in the unknown!

Hello, I'm **Paolo**!

Node.js Technical Steering Committee Member

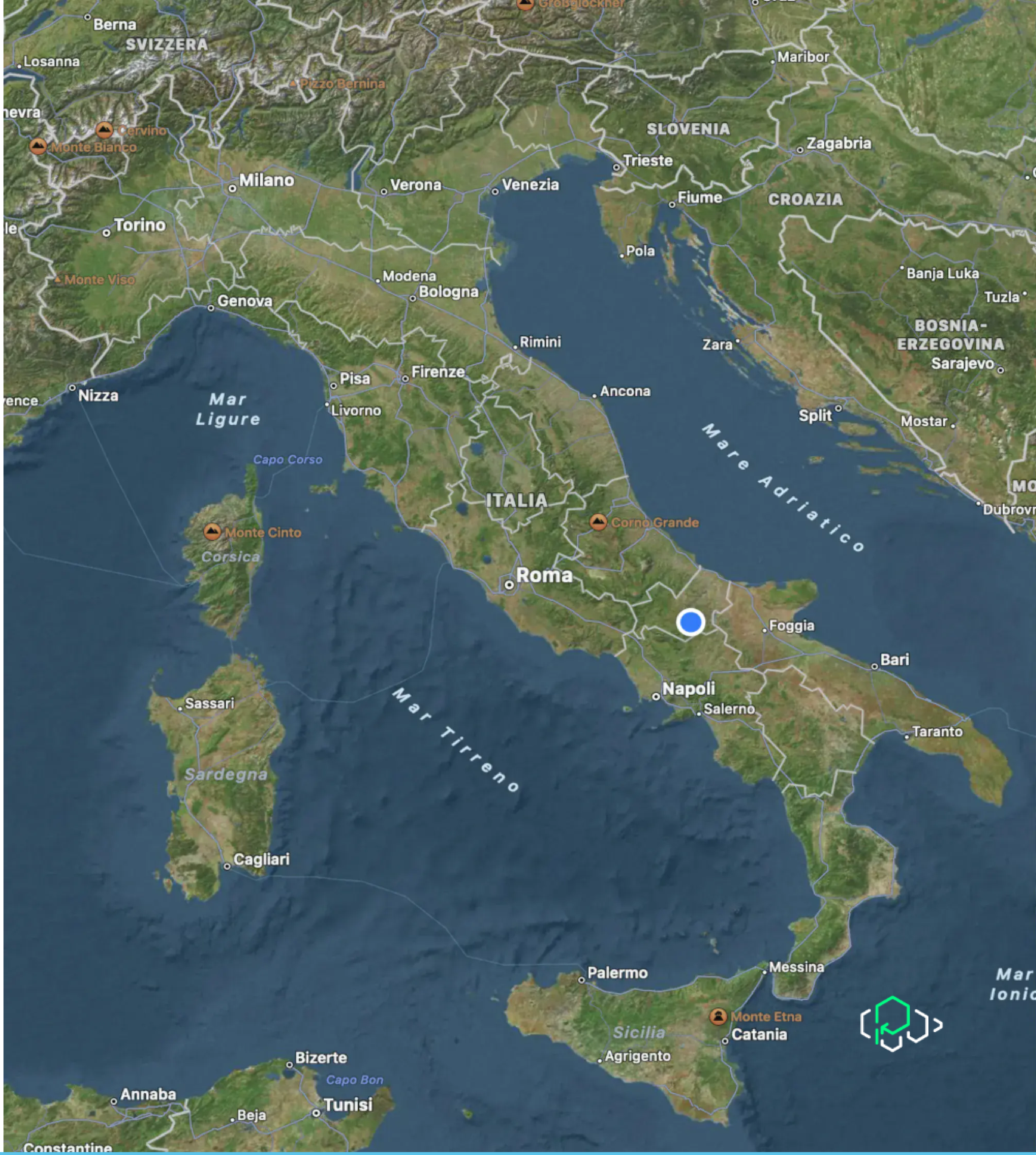
Platformatic Principal Engineer

paoloinsogna.dev

ShogunPanda

p_insogna

pinsogna



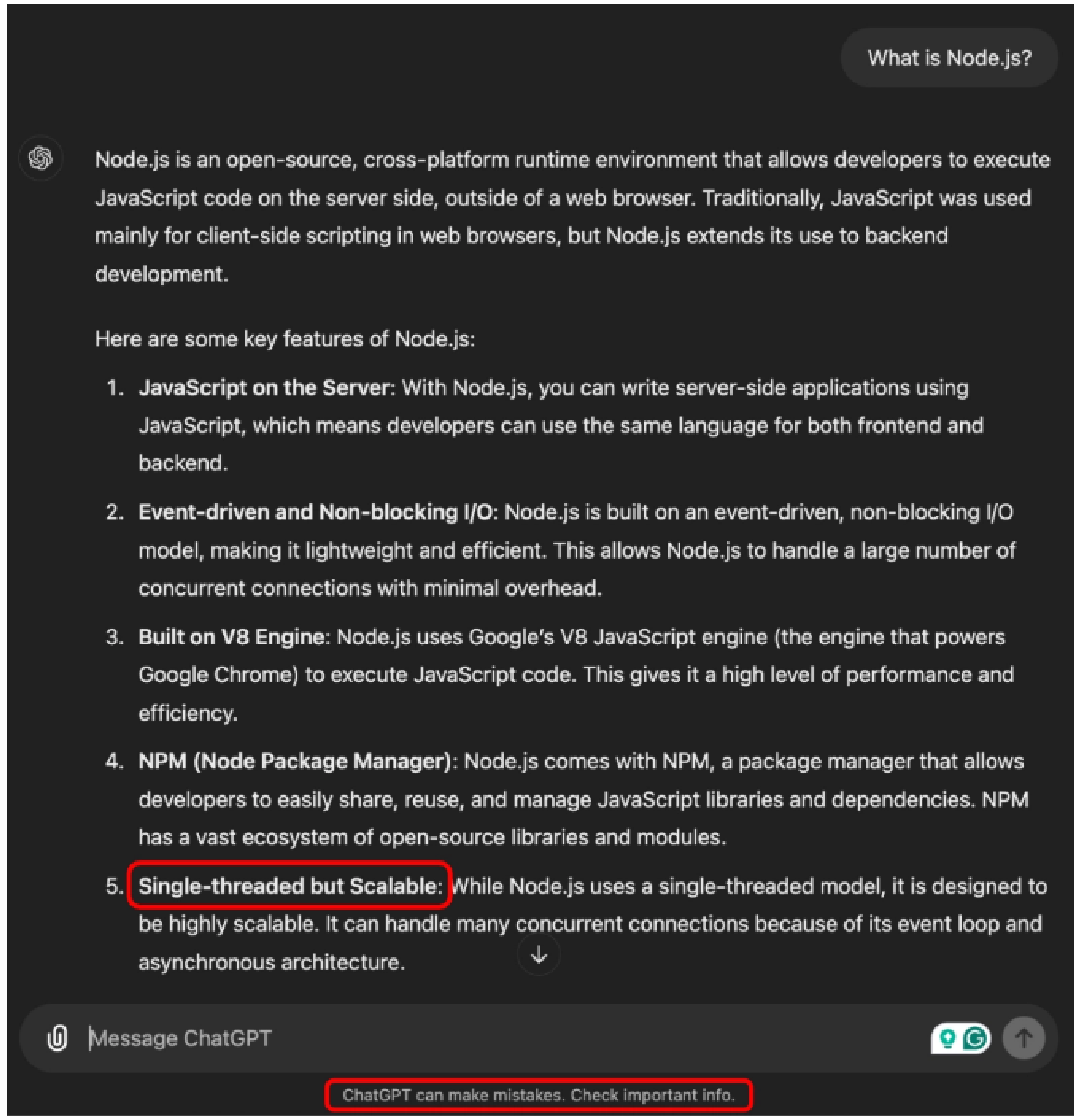
First of all, let's give credits!

This talk is co-authored by Platformatic CTO and friend of mine, the only **Matteo Collina**.

Whatever goes wrong today, please complain directly to him on Twitter!

@matteocollina

Let's start the right way! 🤦‍♂️



What is Node.js?

Node.js is an open-source, cross-platform runtime environment that allows developers to execute JavaScript code on the server side, outside of a web browser. Traditionally, JavaScript was used mainly for client-side scripting in web browsers, but Node.js extends its use to backend development.

Here are some key features of Node.js:

1. **JavaScript on the Server**: With Node.js, you can write server-side applications using JavaScript, which means developers can use the same language for both frontend and backend.
2. **Event-driven and Non-blocking I/O**: Node.js is built on an event-driven, non-blocking I/O model, making it lightweight and efficient. This allows Node.js to handle a large number of concurrent connections with minimal overhead.
3. **Built on V8 Engine**: Node.js uses Google's V8 JavaScript engine (the engine that powers Google Chrome) to execute JavaScript code. This gives it a high level of performance and efficiency.
4. **NPM (Node Package Manager)**: Node.js comes with NPM, a package manager that allows developers to easily share, reuse, and manage JavaScript libraries and dependencies. NPM has a vast ecosystem of open-source libraries and modules.
5. **Single-threaded but Scalable**: While Node.js uses a single-threaded model, it is designed to be highly scalable. It can handle many concurrent connections because of its event loop and asynchronous architecture.

ChatGPT can make mistakes. Check important info.

Node.js is (no longer) single threaded ...

... and it hasn't been for a while now!

2018: “Node.js has threads!”

https://www.youtube.com/watch?v=-ssCzHoUI7M

Worker Thread API

This is supported from Node.js 10.5.0 (June 2018).

Create workers via `worker_threads` **module**
https://nodejs.org/dist/latest-v22.x/docs/api/worker_threads.html

Each thread has an independent event loop
This is crucial to off-load CPU intensive tasks out of the main thread.

How do threads communicate?

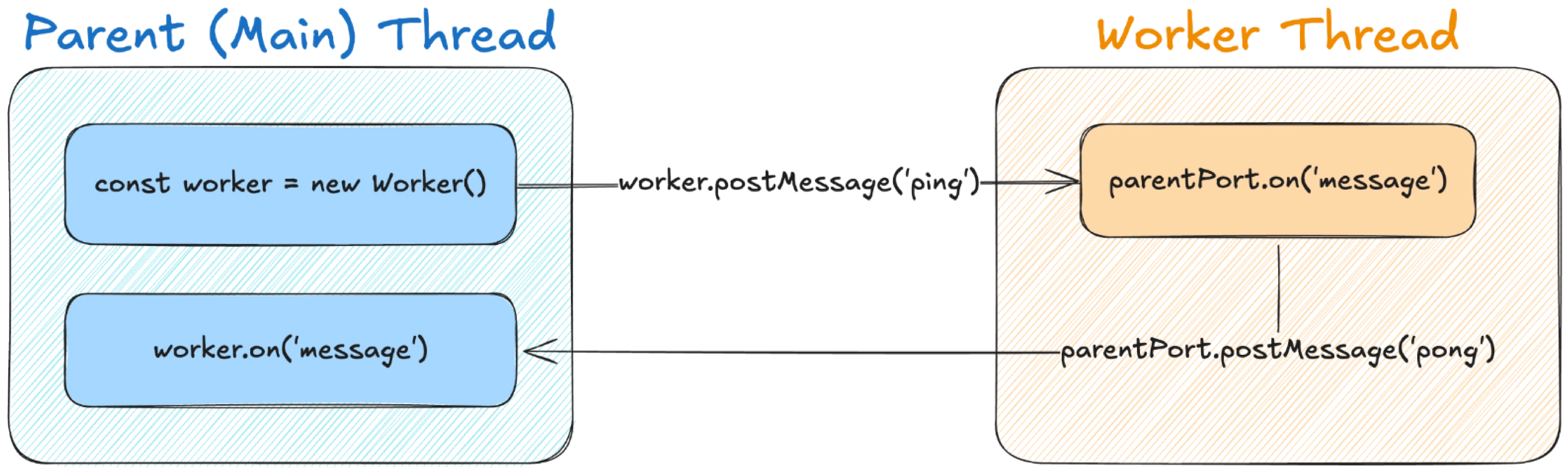
The API is fairly simple and straightforward.
Each thread can communicate with others by sending messages over a `MessagePort`.

Additional channels can be created via the `MessageChannel` API.
It returns a pair of message ports, one meant to be sent to the other thread.

Do you see how far we have gone?

How do threads communicate?

The graph below shows a summary of parent-children thread communication.


Parent (Main) Thread
Worker Thread
const worker = new Worker()
worker.postMessage('ping')
parentPort.on('message')
worker.on('message')
parentPort.postMessage('pong')

BroadcastChannel API

Less known, equally powerful.

This can be used to implement a simple Pub/Sub pattern.

All channels are identified by a global ID and each thread can subscribe to events.

```
import {
  BroadcastChannel,
  isMainThread,
  Worker
} from 'node:worker_threads'

const channel = new BroadcastChannel('main')

if (isMainThread) {
  channel.onmessage = function (event) {
    console.log(event.data) // Will print 10 events
  }

  for (let n = 0; n < 10; n++) {
    new Worker(new URL(import.meta.url))
  }
} else {
  channel.postMessage('ready')
}
```

postMessageToThread

This is only available in Node 22.5.0+.

The API enables for direct thread to thread communication without the need of using `MessageChannel`.

The recipient has to install a handler for the `process.workerMessage` event in order to receive messages.

```
import {
  isMainThread,
  postMessageToThread,
  threadId,
  Worker
} from 'node:worker_threads'

if (isMainThread) {
  for (let n = 0; n < 3; n++) {
    new Worker(new URL(import.meta.url))
  }
} else if (threadId === 1) {
  postMessageToThread(3, { message: 'ready' })
} else {
  process.on('workerMessage', (value, source) => {
    // Will print "1 -> 3: Ready"
    console.log(`${source} -> ${threadId}:`, value)
  })
}
```

How can threads communicate?

Efficient inter-thread RPC

The `node:events` API alow is not suitable for the job.
It is not designed to work between threads.

Let's add promises!
By leveraging the EventEmitter and the Promise API, we can easily build a RPC system.

All you need is an unique identifier and `resolve` method.
The ID is sent to the other thread so that can be used later in a shared event handler.

Example: multithreaded HTTP server

We all know this snippet is more worth than 1000 words.

```
// In the server creation function, define the following
const pending = {}
worker.on('message', message => {
  pending[message.id](message)
})

// In the route handler, define the following
let resolve
const promise = new Promise(_resolve => {
  pending[request.reqId] = _resolve
})

worker.postMessage({id: request.reqId })
const hash = await promise
```

Is all that easy?

Structured Clone

The reality is a bit harsher.

Not all objects can be sent via a MessagePort.
A object cannot contain functions or some native Node.js object (as `net.Socket`).

The object will be cloned according to the Structured Clone Algorithm.
Check the documentation on MDN!

Transferable (1/2)

You can't always clone everything.

Some objects, usually the ones backed by a C++ representation, can only be **moved** between threads.

These objects are called **transferrable** and thus must be explicitly listed in the second argument of `postMessage`.

Transferable (2/2)

Some example of transferable objects are:

- `MessagePort`
- `CryptoKey`
- `FileHandle`
- `WebStream` (`ReadableStream` and `WritableStream`)

The full list can be found in the **MessagePort documentation**.

Ready for another dive?

Piscina

It is a powerful NPM module which provides an easy to use API.

A Piscina instance dynamically adjusts the number of workers to the maximum number of parallelism available on the machine.

All the messaging and synchronization is managed so you focus on the core logic.

https://www.npmjs.com/package/piscina

How to use Piscina

How easy can multithreading be?

```javascript
// Main thread

import { Piscina } from 'piscina'

const piscina = new Piscina({
  filename: new URL(
    './worker.mjs', import.meta.url
  ).href
})

const result = await piscina.run({ a: 4, b: 6 })
console.log(result) // Prints 10
```

```javascript
// Worker thread

export default ({ a, b }) => {
  return a + b
}
```

Are we done?

Do you know what you can use Worker Threads for?

“It's not who you are underneath,
it’s what you do that defines you”

“Everything is impossible until somebody does it”

How can a Promise be invoked synchronously?

You can use Worker Threads to invoke an async function... **synchronously**!

```
import { setTimeout as sleep } from 'node:timers/promises'

async function echo (value) {
  await sleep(1000)
  return value
}
```

everysync (1/2)

A tiny utility to expose an asynchronous API via a worker thread... synchronously!

https://github.com/mcollina/everysync

```
import { join } from 'node:path'
import { strictEqual } from 'node:assert'
import { Worker } from 'node:worker_threads'
import { makeSync } from 'everysync'

const buffer = new SharedArrayBuffer(1024, {
  maxByteLength: 64 * 1024 * 1024,
})
const worker = new Worker(join(__dirname, 'echo.mjs'), {
  workerData: { data: buffer },
})
const api = makeSync(buffer)

assert.strictEqual(api.echo(42), 42)
worker.terminate()
```

everysync (2/2)

This is all you need to expose an API from a thread.

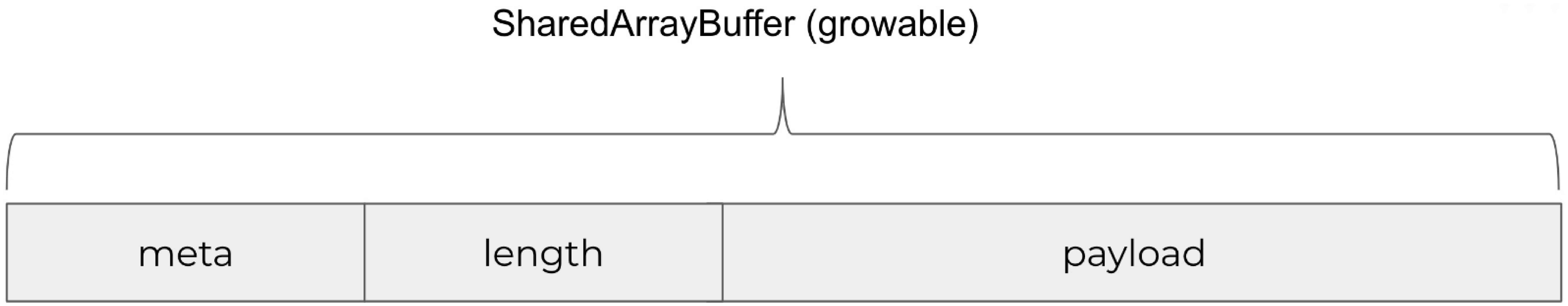
```
import { wire } from 'everysync'
import { workerData } from 'node:worker_threads'
import { setTimeout } from 'node:timers/promises'

wire(workerData.data, {
  async echo(value) {
      await setTimeout(1000)
      return value
  }
})

// Keep the event loop alive
setInterval(() => {}, 100000)
```

How?

`Atomics.waitAsync` **and** `SharedArrayBuffer`

`Atomics.waitAsync` is used to synchronize the main thread with the worker thread in meta.

Payload contains the message to be sent, and length its is size.



Use the same mechanism of `postMessage`

https://github.com/mcollina/everysync/blob/main/lib/objects.js

```
import { serialize, deserialize } from 'node:v8'

function read(buffer, byteOffset = 0) {
  const view = new DataView(buffer, byteOffset)
  const length = view.getUint32(0, true)
  const object = deserialize(new Uint8Array(buffer, byteOffset + 4, length))
  return object
}

function write(buffer, object, byteOffset = 0) {
  const data = serialize(object)

  if (buffer.byteLength < data.byteLength + 4 + byteOffset) {
    if (!buffer.growable) {
      throw new Error('Buffer is not growable')
    }

    buffer.grow(data.byteLength + 4 + byteOffset)
  }

  const view = new DataView(buffer, byteOffset)
  view.setUint32(0, data.byteLength, true)
  new Uint8Array(buffer, byteOffset + 4).set(data)
}
```

Calling a method to the worker thread

All happens thanks to the `SharedArrayBuffer`.

```
const OFFSET = 64
// * 0: writing from worker, reading from main
const TO_WORKER = 0
// * 1: writing from main, reading from worker
const TO_MAIN = 1

const data = new SharedArrayBuffer(1024, {
  maxByteLength: 64 * 1024 * 1024,
})

const metaView = new Int32Array(data)
write(data, { key: 'echo', args: "42" }, OFFSET)
Atomics.store(metaView, TO_MAIN, 1)
Atomics.notify(metaView, TO_MAIN, 1)

const res = Atomics.wait(metaView, TO_WORKER, 0, timeout)
Atomics.store(metaView, TO_WORKER, 0)
if (res === 'ok') {
  const obj = read(data, OFFSET)
  console.log('result', obj)
} else {
  throw new Error(`The response timed out after ${timeout}ms`)
}
```

Receiving the calls using Atomics.waitAsync

Don’t forget to also use `Atomics.notify`.

```js
import { workerData } from 'node:worker_threads'
import { setTimeout as sleep } from 'node:timers/promises'
const data = workerData.data
const metaView = new Int32Array(data)
// ...

const obj = { echo }

while (true) {
  const waitAsync = Atomics.waitAsync(metaView, TO_MAIN, 0)
  const res = await waitAsync.value
  Atomics.store(metaView, TO_MAIN, 0)

  if (res === 'ok') {
    const { key, args } = read(data, OFFSET)
    const result = await obj[key](...args)
    write(data, result, OFFSET)
    Atomics.store(metaView, TO_WORKER, 1)
    Atomics.notify(metaView, TO_WORKER, 1)
  }
}
```

Why this is useful?

Pino uses this technique to support async transports and flush the logs on exit.

https://getpino.io

Loader hooks

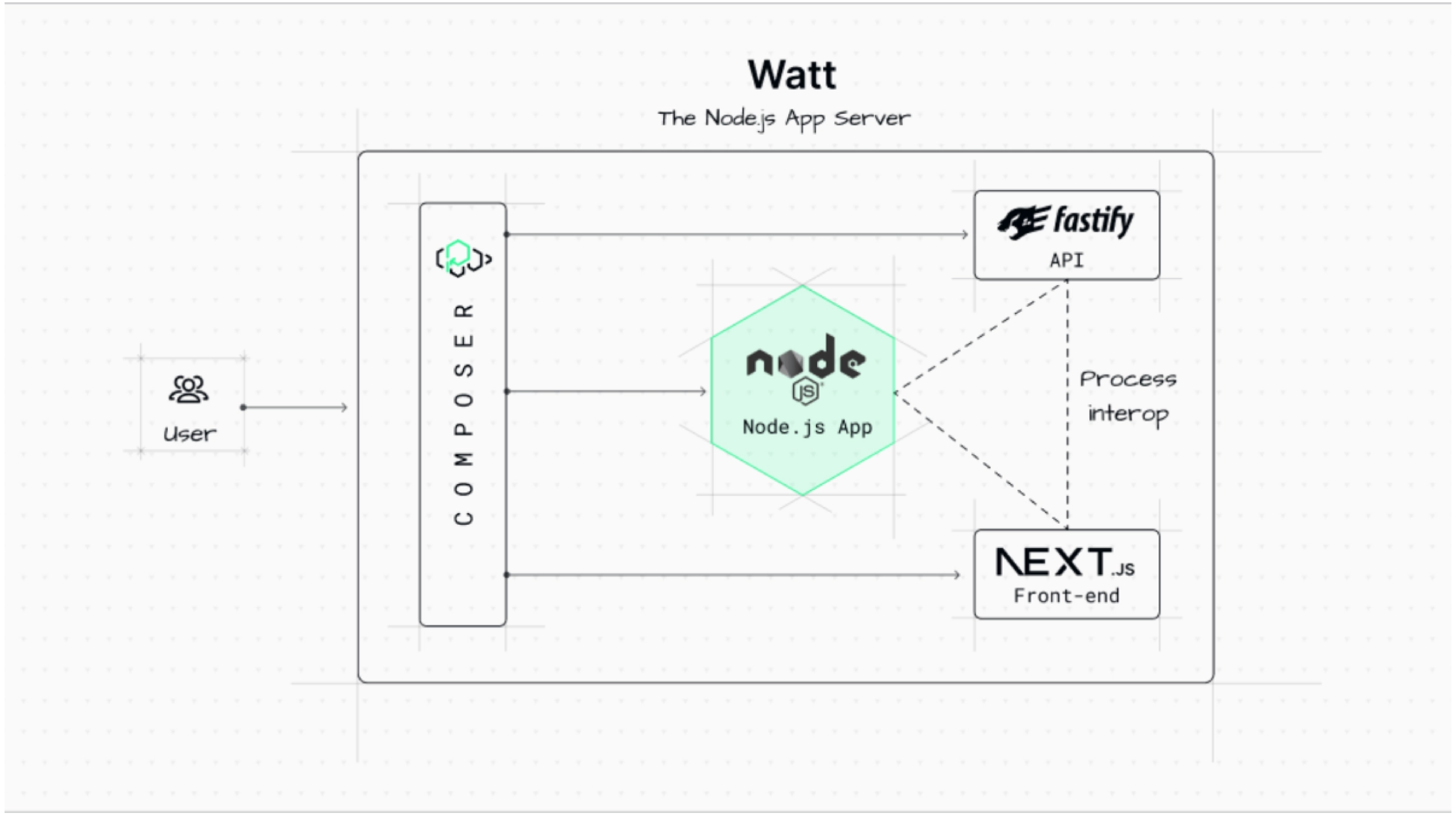
Node.js supports loading hooks via the `node:module` module, which allows to customize the resolving and the loading of any file or module, including Node.js internals.

While the `import` or `require` statements are still synchronous, all the works under the hood happens in an asynchronous way thanks to the use of an internal separate worker thread, with a mechanism similar to everysync.

The multithreading architecture is the foundation of features like `require(esm)` or `-experimental-strip-types`.

Are we finally done?

Introducing Watt, the Node.js application server



Watt

Watt (**https://platformatic.dev/watt**) abstracts away time-consuming tasks like monitoring, logging and tracing all while supporting full stack applications.

It allows you to run multiple Node.js services within the same process with inter-threading communication over HTTP.

Each service runs in its worker thread so they cannot interfere with each other.

How do you configure Watt?

An easy to understand configuration file is all you need. Create it with `wattpm init`.

```json
{
  "$schema": "https://schemas.platformatic.dev/wattpm/2.44.0.json",
  "server": {
    "hostname": "127.0.0.1",
    "port": 3000
  },
  "logger": {
    "level": "error"
  },
  "autoload": {
    "path": "web"
  },
  "watch": true
}
```

How do you create a service?

Conventions over configuration. **Ring a bell?**

Create a folder named after the service in the `web` folder then create a `package.json` file with some values like in the snippet below.

```json
{
  "private": true,
  "type": "module",
  "main": "index.js",
  "dependencies": {
    "@platformatic/node": "^2.44.0"
  }
}
```

Configure your service

Create a new watt.json in the service folder.

You can use **`wattpm import`** to get one very easily. It generates the following file.

```
shogun@panda:~/example$ wattpm import
shogun@panda:~/example$ cat watt.json
{
  "$schema": "https://schemas.platformatic.dev/@platformatic/node/2.44.0.json"
}
```

How do you write a service?

1 **Create a simple Node.js application**

Export a `build` or `create` function which returns a supported HTTP application.

2 **You can omit `listen`.**

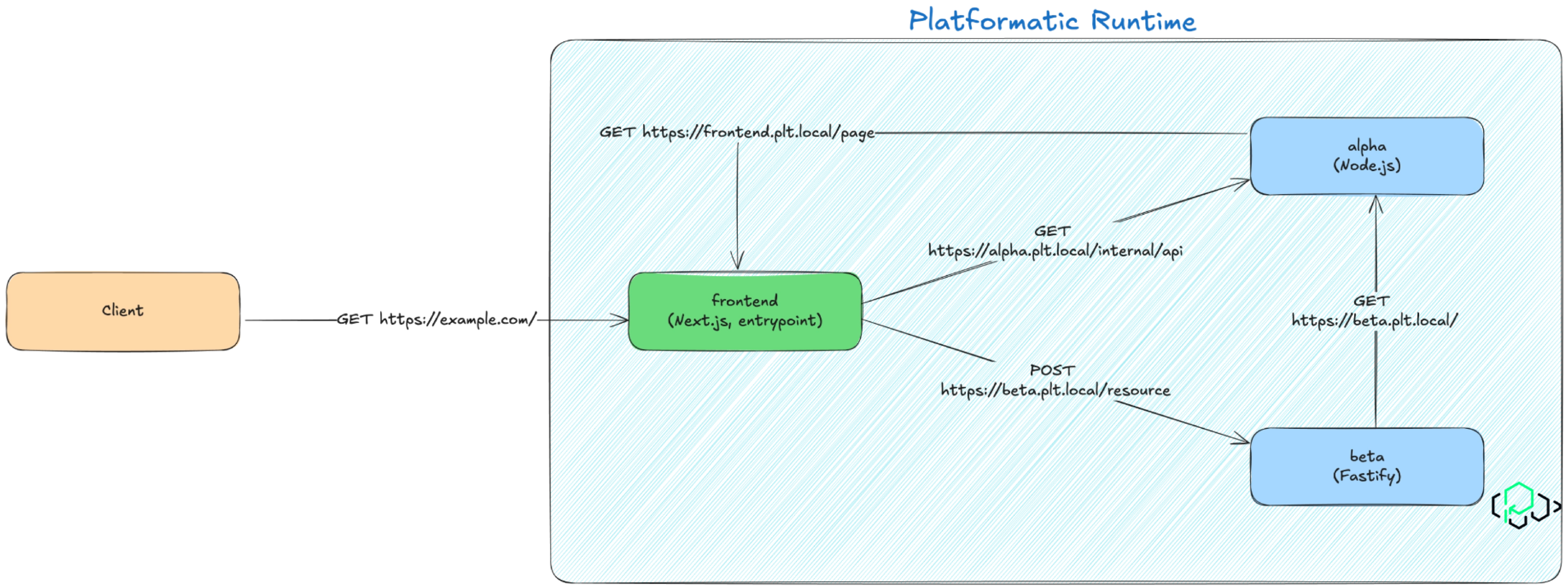
Watt will automatically invoke listen for entrypoint only. Other services are not exposed.

3 **Use the mesh network via `fetch`.**

Each service is internally reachable on the `http://[SERVICE].plt.local` domain.

Network-less HTTP

The Platformatic mesh network. Our secret sauce.


Platformatic Runtime
Client
GET https://example.com/
frontend (Next.js, entrypoint)
GET https://frontend.plt.local/page
alpha (Node.js)
GET https://alpha.plt.local/internal/api
POST https://beta.plt.local/resource
beta (Fastify)
GET https://beta.plt.local/

Example: a service which invokes another service

Like earlier, we all know this snippet is more worth than 1000 words.

```
import fastify from 'fastify'

export function create() {
  const app = fastify({})

  app.get('/fast', async () => {
    return { time: Date.now() }
  })

  app.get('/slow', async () => {
    const response = await fetch('http://worker.plt.local/hash')
    return response.json()
  })

  return app
}
```

INTERNATIONAL KIDS CODING BOOTCAMP

International Kids Coding Bootcamp

GoFundMe campaign

We are running a GoFundMe campaign to support the Code Their Future initiative.

https://www.gofundme.com/f/code-their-future-unlocking-opportunities-for-underserved-k

One last thing™

“Keep your face always toward the sunshine and shadows will fall behind you.”

Walt Whitman

Thank you!

Paolo Insogna
Node.js TSC, Principal Engineer

@p_insogna
paolo.insogna@platformatic.dev

Platformatic