A new way to profile Node.js

Matteo Collina

NearForm

Maximum number of servers

sales traffic dropping

angry people are angry

Why is it slow?

because bottleneck

Why is it slow?

The bottleneck is internal

The Node.js process
is on fire

The bottleneck is external

Something else
is on fire

Where is the bottleneck?

Finding Bottlenecks

Simulating Load

Finding Bottlenecks

Diagnostics

Clinic.js

```
$ npm install -g clinic
```

Clinic.js

Clinic Doctor

Clinic Bubbleprof

Clinic Flame

Clinic Doctor

Collects metrics by injecting probes

Assesses health with heuristics

Creates recommendations

Doctor metrics

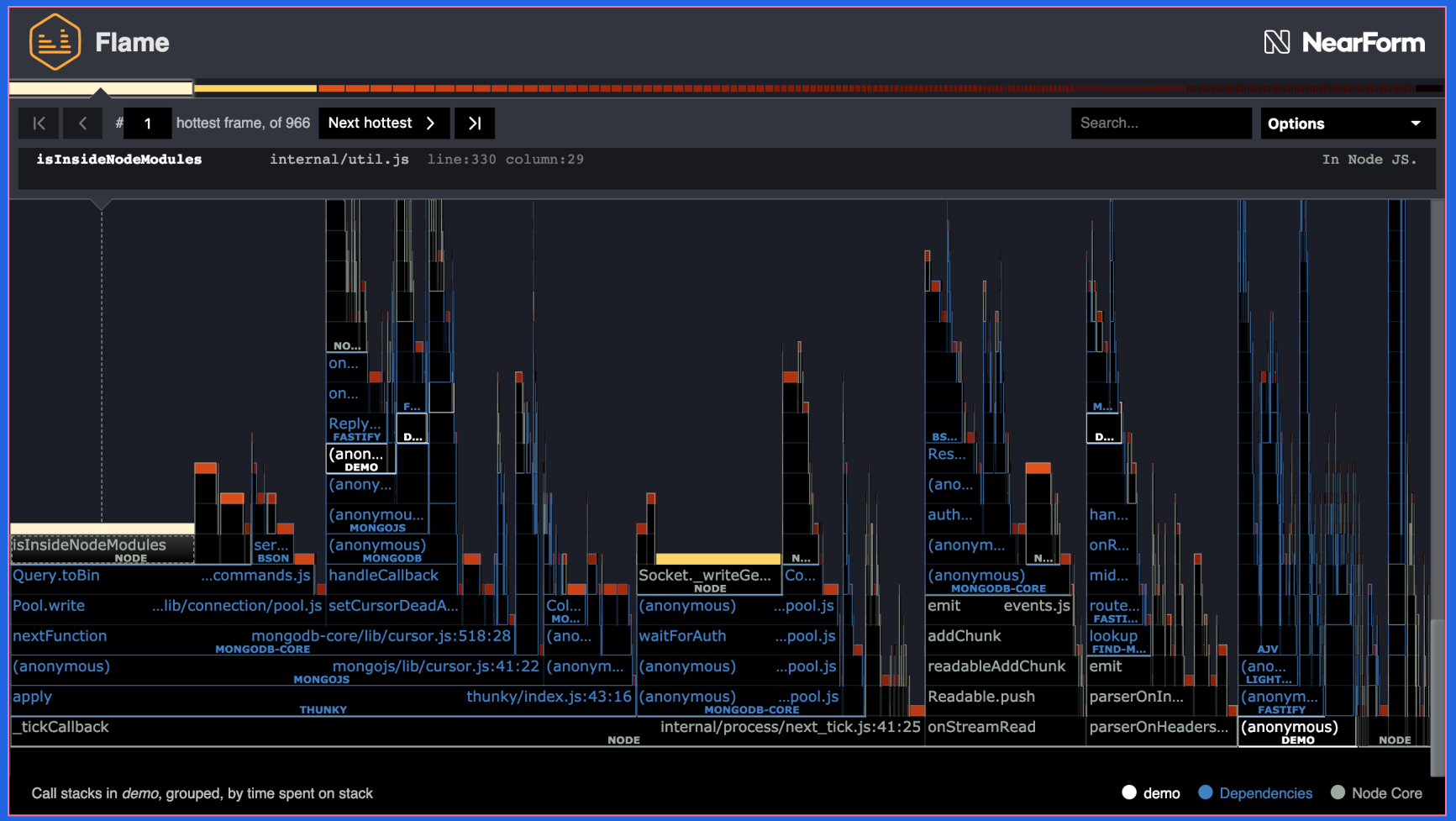
Clinic Flame

Collects metrics by CPU sampling

Tracks top-of-stack frequency

Creates flame graphs

Flame graphs



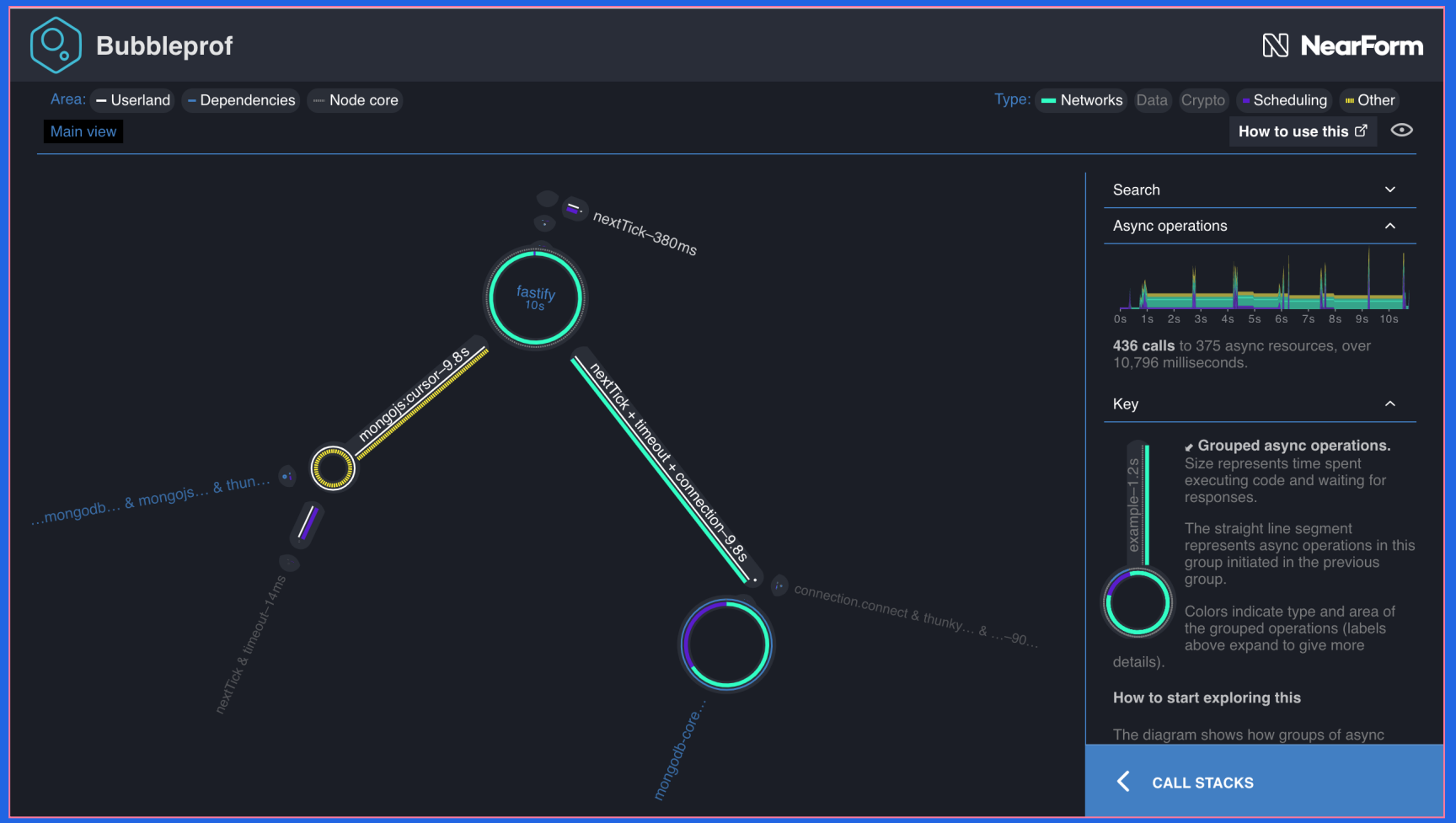
Clinic Bubbleprof

Collects metrics using async_hooks

Tracks latency between operations

Creates bubble graphs

Bubble graphs



Where is the bottleneck?

Clinic Flame

For internal bottlenecks

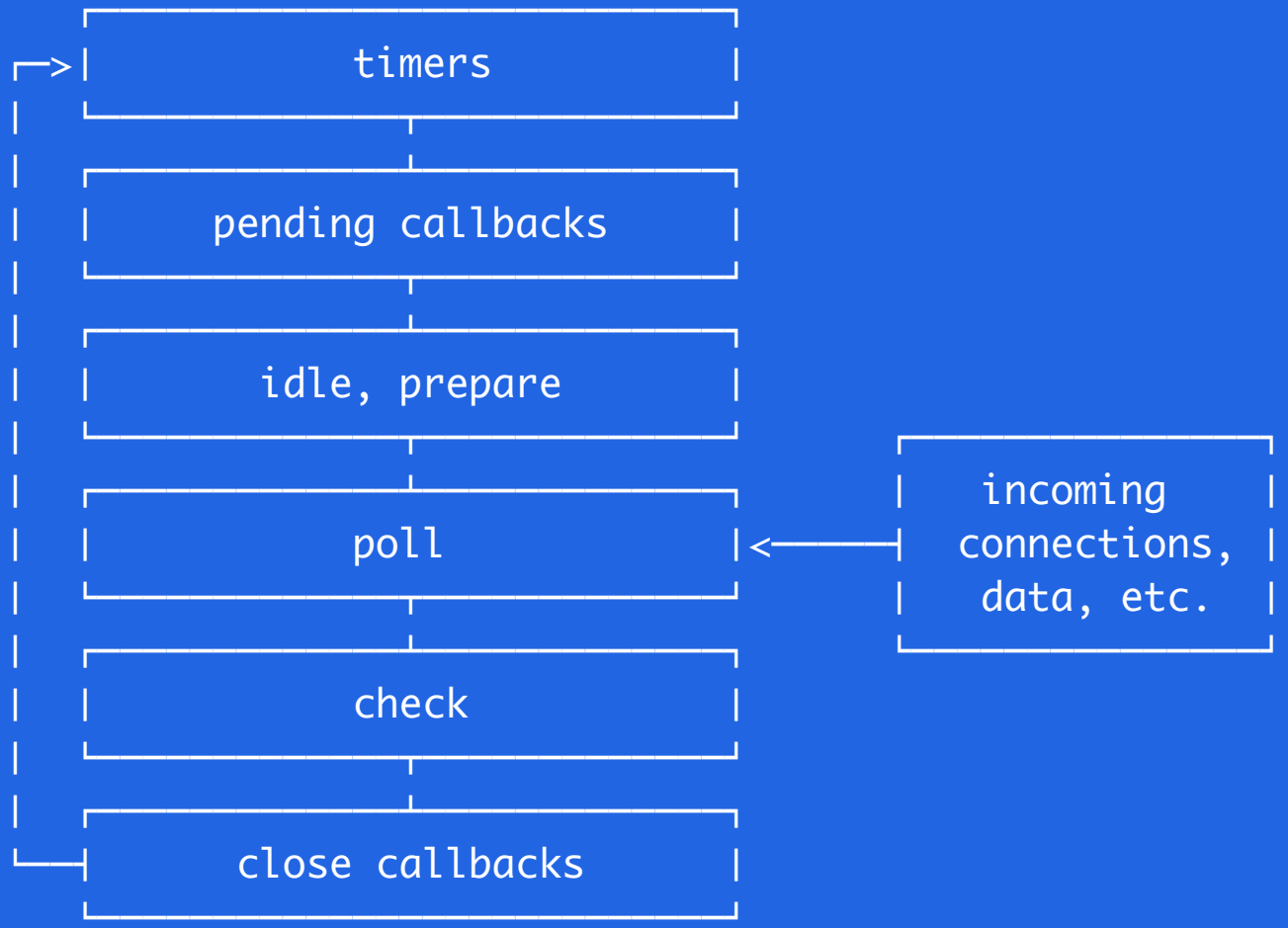
Clinic Bubbleprof

For external bottlenecks

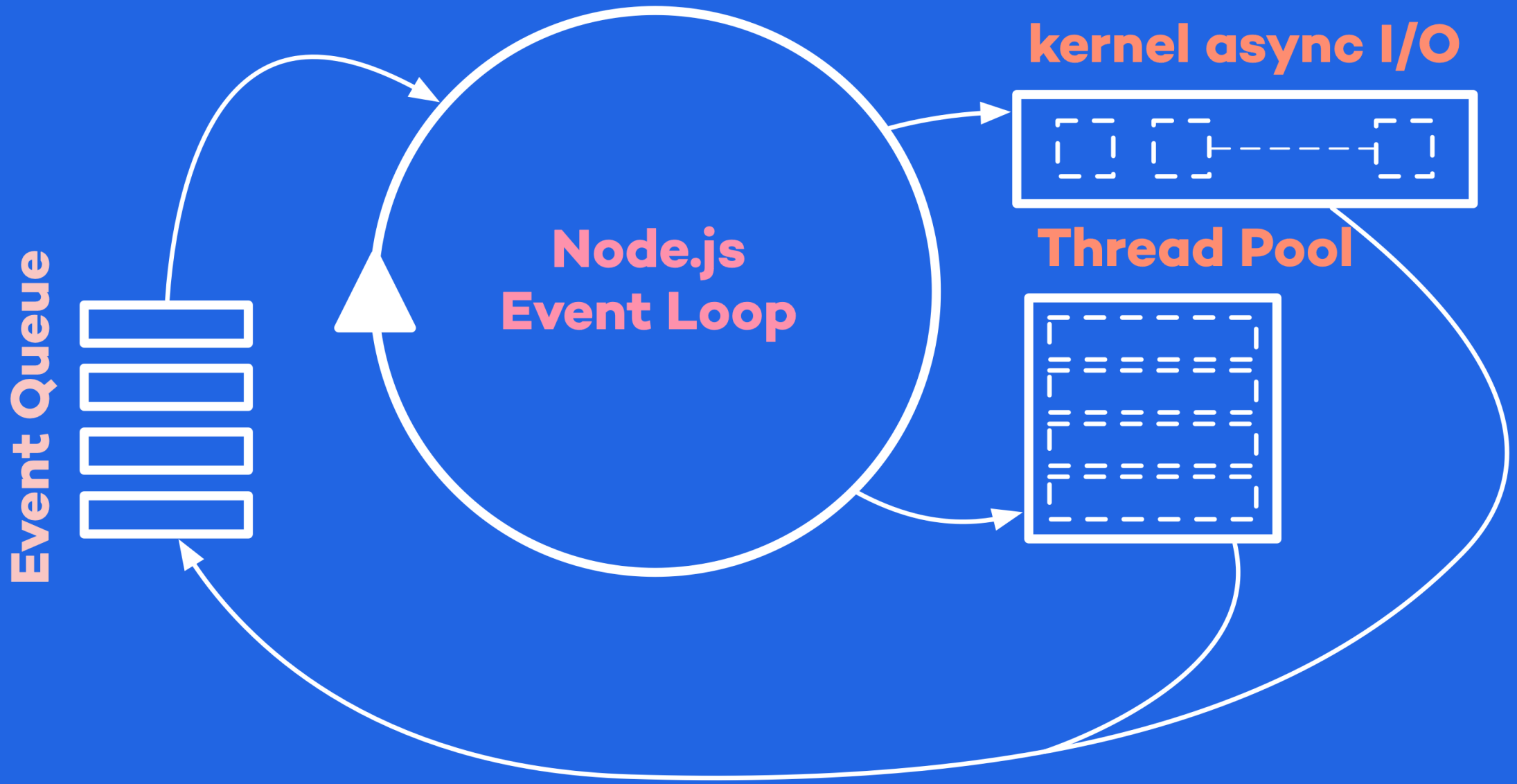
Live Hack

How can we improve the performance of our Node.js apps?

The Event Loop



Source: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/


kernel async I/O
Thread Pool
Node.js
Event Loop
Event Queue

The life of an event

1. JS adds a *function* as a listener for an I/O event
2. The *I/O event* happens
3. The specified function is *called*

In Node.js, there is no parallelism of function execution.

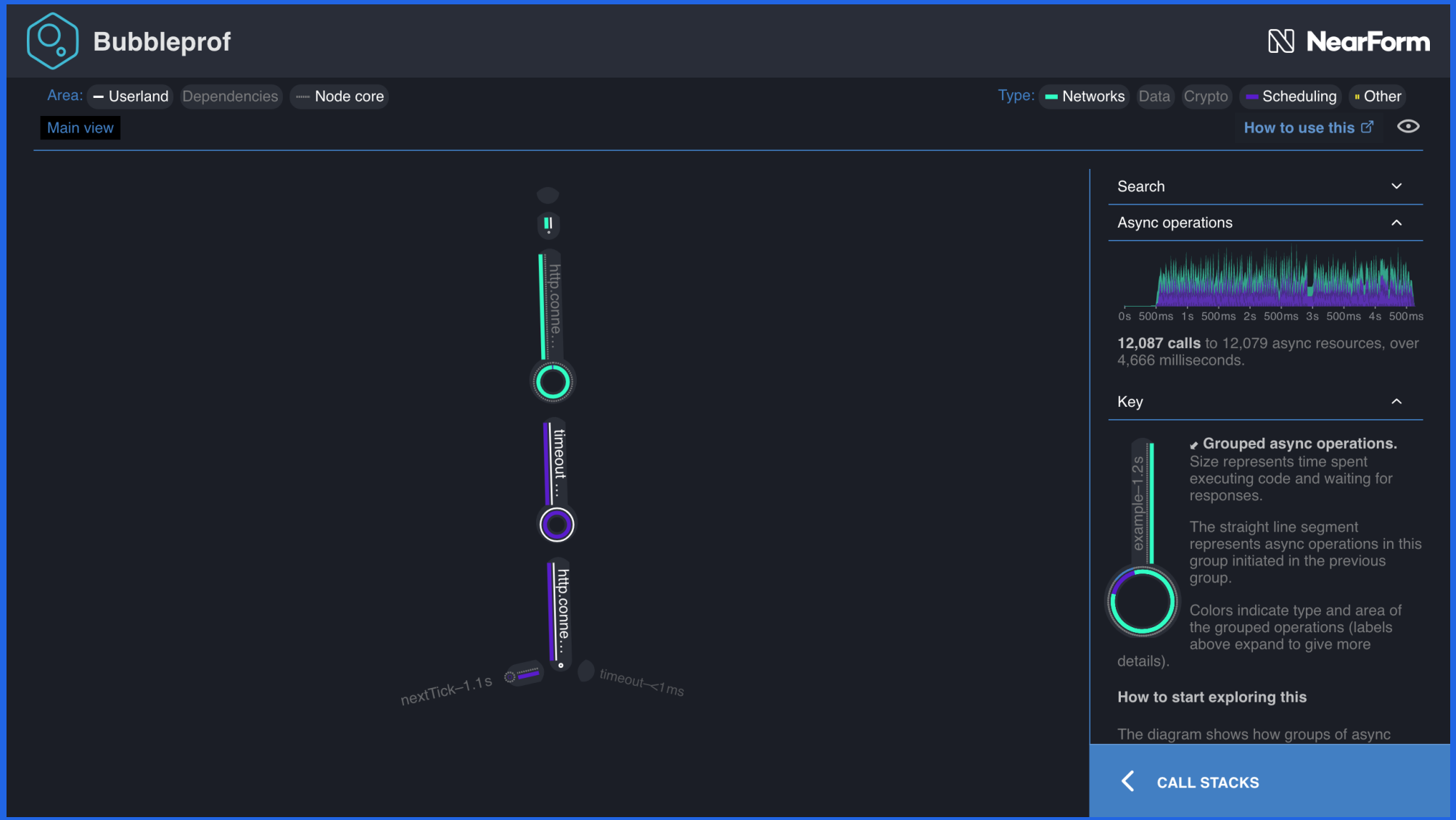
`nextTick`, `Promises`, `setImmediate`

1. `nextTicks` are *always* executed *before* `Promises` and other *I/O events*.
2. `Promises` are *executed synchronously* and *resolved asynchronously*, before any other I/O events.
3. `setImmediate` exercise the same *flow* of *I/O events*.

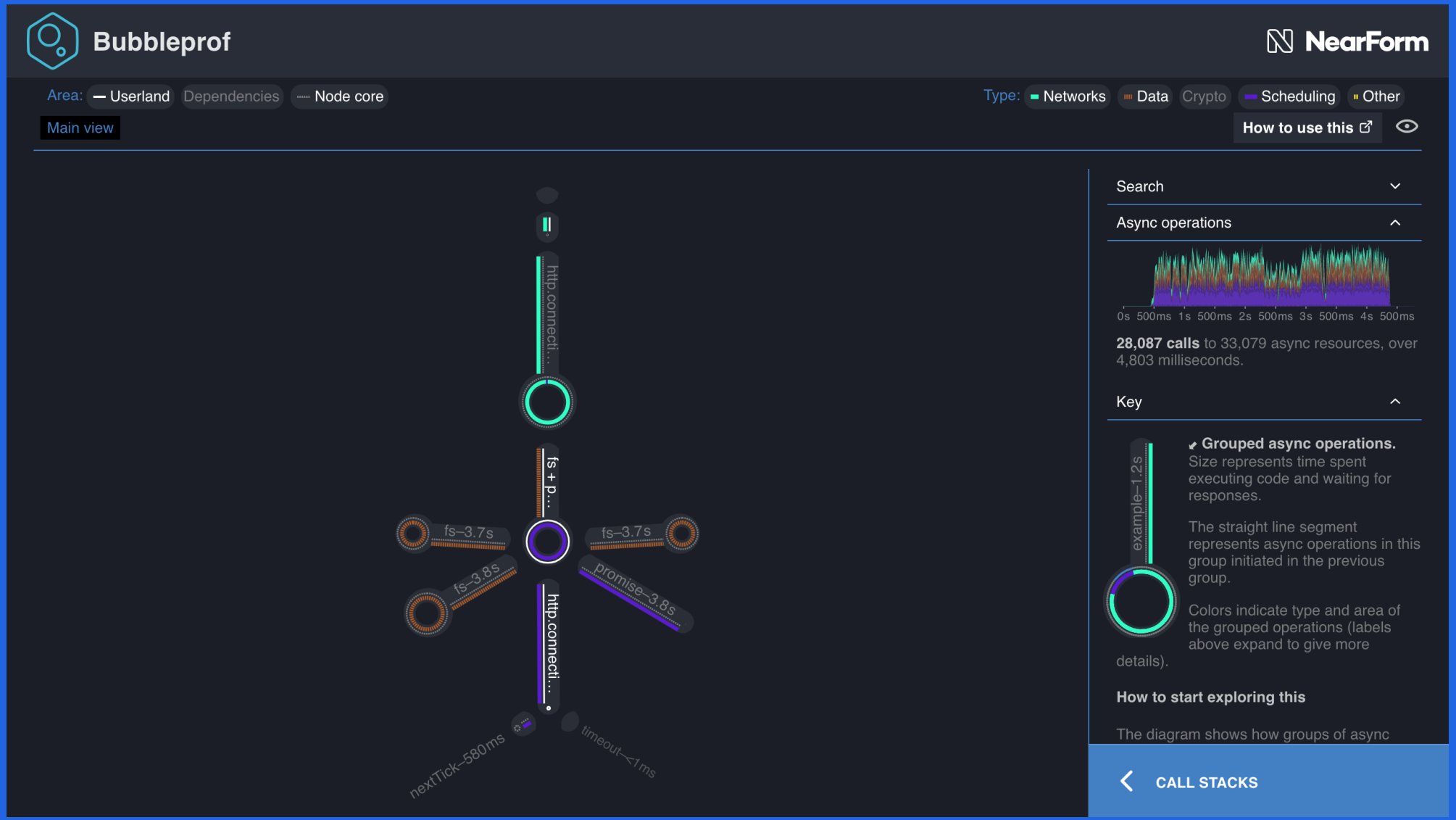
The hardest concept in Node.js is to know when a chunk of code is running relative to another.

Clinicjs can help you in understanding how your Node.js application works

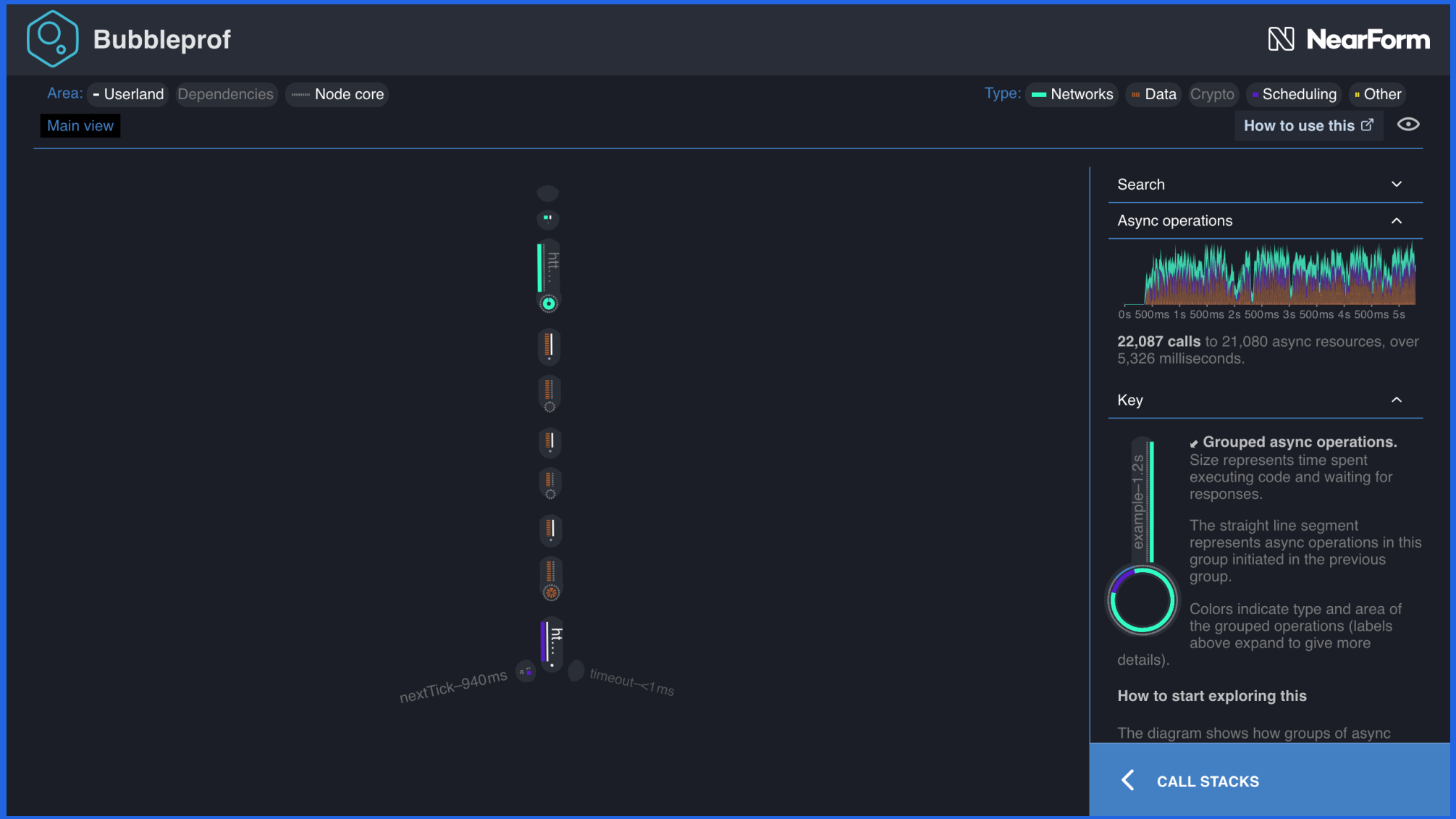
```
const http = require('http')
const { promisify } = require('util')
const sleep = promisify(setTimeout)

http.createServer(function handle (req, res) => {
  sleep(20).then(() => {
    res.end('hello world')
  }, (err) => {
    res.statusCode = 500
    res.end(JSON.stringify(err))
  })
}).listen(3000)
```


Bubbleprof
NearForm
Area: Userland Dependencies Node core
Type: Networks Data Crypto Scheduling Other
Main view
How to use this
http.conne...
timeout ...
http.conne...
nextTick–1.1 s
timeout–<1 ms
Search
Async operations
0s 500ms 1s 500ms 2s 500ms 3s 500ms 4s 500ms
12,087 calls to 12,079 async resources, over 4,666 milliseconds.
Key
example–1.2 s
Grouped async operations. Size represents time spent executing code and waiting for responses.
The straight line segment represents async operations in this group initiated in the previous group.
Colors indicate type and area of the grouped operations (labels above expand to give more details).
How to start exploring this
The diagram shows how groups of async
CALL STACKS

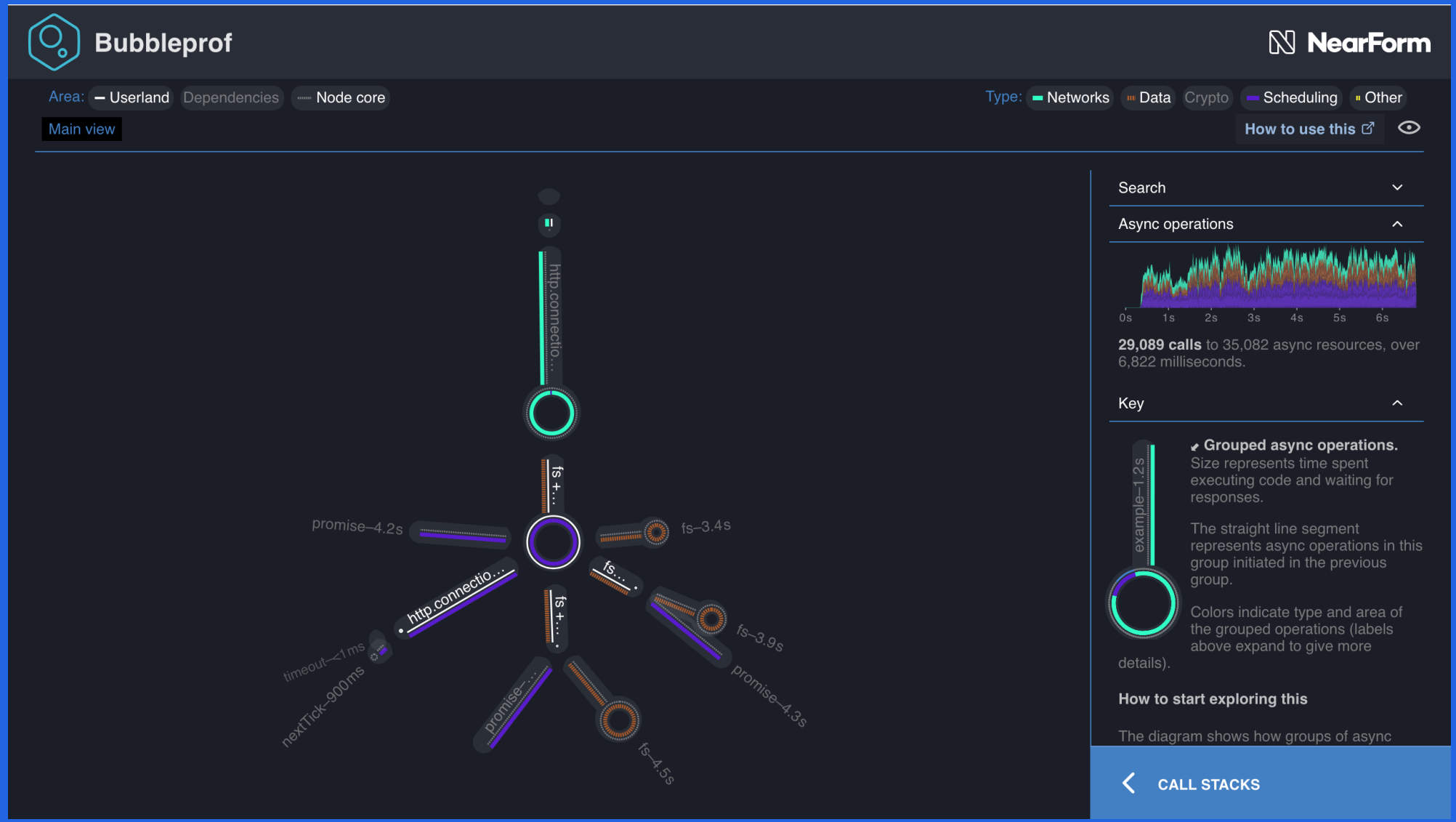
```
const http = require('http')
const { promisify } = require('util')
const sleep = promisify(setTimeout)

async function something (req, res) {
  await sleep(20)
  res.end('hello world')
}

http.createServer(function handle (req, res) {
  something(req, res).catch((err) => {
    res.statusCode = 500
    res.end(JSON.stringify(err))
  })
}).listen(3000)
```


Bubbleprof
NearForm
Area: Userland Dependencies Node core
Type: Networks Data Crypto Scheduling Other
Main view
How to use this
http.connecti...
fs + p...
fs–3.7s
fs–3.7s
fs–3.8s
promise–3.8s
http.connecti...
nextTick–580ms
timeout–<1ms
Search
Async operations
0s 500ms 1s 500ms 2s 500ms 3s 500ms 4s 500ms
28,087 calls to 33,079 async resources, over 4,803 milliseconds.
Key
example–1.2s
Grouped async operations.
Size represents time spent executing code and waiting for responses.
The straight line segment represents async operations in this group initiated in the previous group.
Colors indicate type and area of the grouped operations (labels above expand to give more details).
How to start exploring this
The diagram shows how groups of async
CALL STACKS

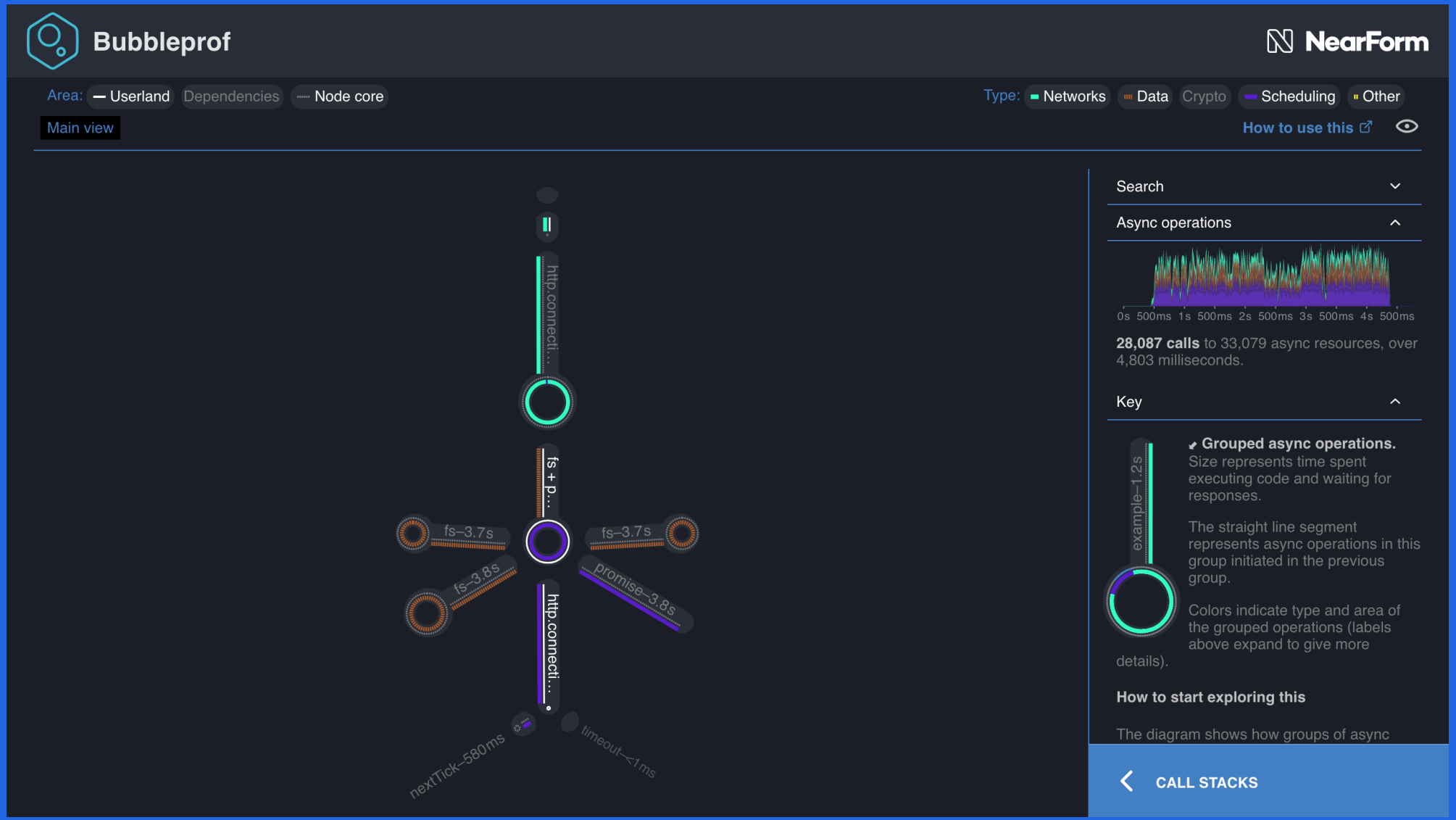
```
const http = require('http')
const fs = require('fs')

http.createServer(function f1 (req, res) {
  fs.readFile(__filename, function f2 (err, buf1) {
    if (err) throw err
    fs.readFile(__filename, function f3 (err, buf2) {
      if (err) throw err
      fs.readFile(__filename, function f4 (err, buf3) {
        if (err) throw err
        res.end(Buffer.concat([buf1, buf2, buf3]))
      })
    })
  })
}).listen(3000)
```


Bubbleprof
NearForm
Area: Userland Dependencies Node core
Type: Networks Data Crypto Scheduling Other
Main view
How to use this
htt...
ht...
nextTick–940ms
timeout–<1ms
Search
Async operations
0s 500ms 1s 500ms 2s 500ms 3s 500ms 4s 500ms 5s
22,087 calls to 21,080 async resources, over 5,326 milliseconds.
Key
example–1.2s
Grouped async operations. Size represents time spent executing code and waiting for responses.
The straight line segment represents async operations in this group initiated in the previous group.
Colors indicate type and area of the grouped operations (labels above expand to give more details).
How to start exploring this
The diagram shows how groups of async
CALL STACKS

```
const http = require('http')
const fs = require('fs')
const { promisify } = require('util')
const readFile = promisify(fs.readFile)

async function handle (req, res) {
  const a = await readFile(__filename)
  const b = await readFile(__filename)
  const c = await readFile(__filename)

  res.end(Buffer.concat([a, b, c]))
}

http.createServer(function (req, res) {
  handle(req, res).catch((err) => {
    res.statusCode = 500
    res.end('kaboom')
  })
}).listen(3000)
```


Bubbleprof
NearForm
Area: Userland Dependencies Node core
Type: Networks Data Crypto Scheduling Other
Main view
How to use this
http.connectio...
fs +...
promise–4.2s
fs–3.4s
fs...
http.connectio...
fs–3.9s
promise–4.3s
timeout–<1 ms
nextTick–900 ms
promise-...
fs–4.5s
Search
Async operations
0s 1s 2s 3s 4s 5s 6s
29,089 calls to 35,082 async resources, over 6,822 milliseconds.
Key
example–1.2s
Grouped async operations.
Size represents time spent executing code and waiting for responses.
The straight line segment represents async operations in this group initiated in the previous group.
Colors indicate type and area of the grouped operations (labels above expand to give more details).
How to start exploring this
The diagram shows how groups of async
CALL STACKS

```
const http = require('http')
const fs = require('fs')
const { promisify } = require('util')
const readFile = promisify(fs.readFile)

async function handle (req, res) {
  res.end(Buffer.concat(await Promise.all([
    readFile(__filename),
    readFile(__filename),
    readFile(__filename)
  ])))
}

http.createServer(function (req, res) {
  handle(req, res).catch((err) => {
    res.statusCode = 500
    res.end('kaboom')
  })
}).listen(3000)
```


Bubbleprof
NearForm
Area: Userland Dependencies Node core
Type: Networks Data Crypto Scheduling Other
Main view
How to use this
http.connecti…
fs + p…
fs–3.7s
fs–3.7s
fs–3.8s
promise–3.8s
http.connecti…
nextTick–580ms
timeout–<1ms
Search
Async operations
0s 500ms 1s 500ms 2s 500ms 3s 500ms 4s 500ms
28,087 calls to 33,079 async resources, over 4,803 milliseconds.
Key
example–1.2s
Grouped async operations.
Size represents time spent executing code and waiting for responses.
The straight line segment represents async operations in this group initiated in the previous group.
Colors indicate type and area of the grouped operations (labels above expand to give more details).
How to start exploring this
The diagram shows how groups of async
CALL STACKS

Performance Considerations

As a result of a slow I/O operation, your application increase the amount of concurrent tasks.

A huge amount of concurrent tasks increase the memory consumption of your application.

An increase in memory consumption increase the amount of work the garbage collector (GC) needs to do on our CPU.

Under high load, the GC will steal CPU cycles from our JavaScript critical path.

Therefore, latency and throughput are connected

Parting Words

Set quantifiable performance goals

> The application should have a response time of 200ms or less in the 99th percentile at 100 concurrent requests per server.

Choose fast libraries

Pino

High speed logging library

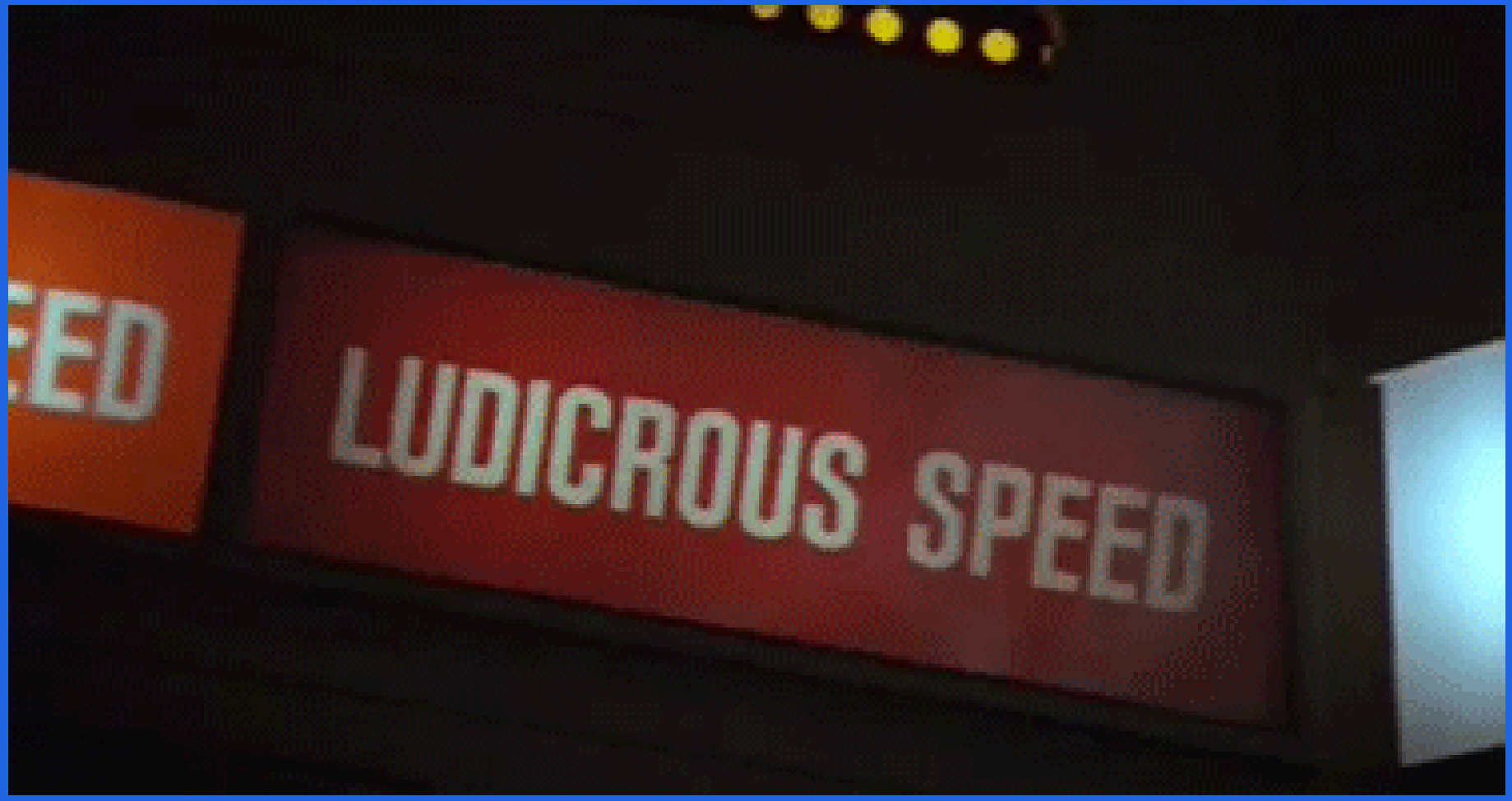
Fastify

High speed web framework

Beware of the rabbit hole

- It is not uncommon for 80% of effort to be in the final 20% of optimization work
- Find out what fast enough is for your given business context
- Remember to balance the cost of your time against savings and other business gains

You don't always have to reach..



Do you need help with your Node.js application?

Questions?

@matteocollina

Thanks

@matteocollina