

Platformatic

View online

Download PDF

Why Node.js needs an application server

Paolo Insogna
Node.js TSC, Principal Engineer

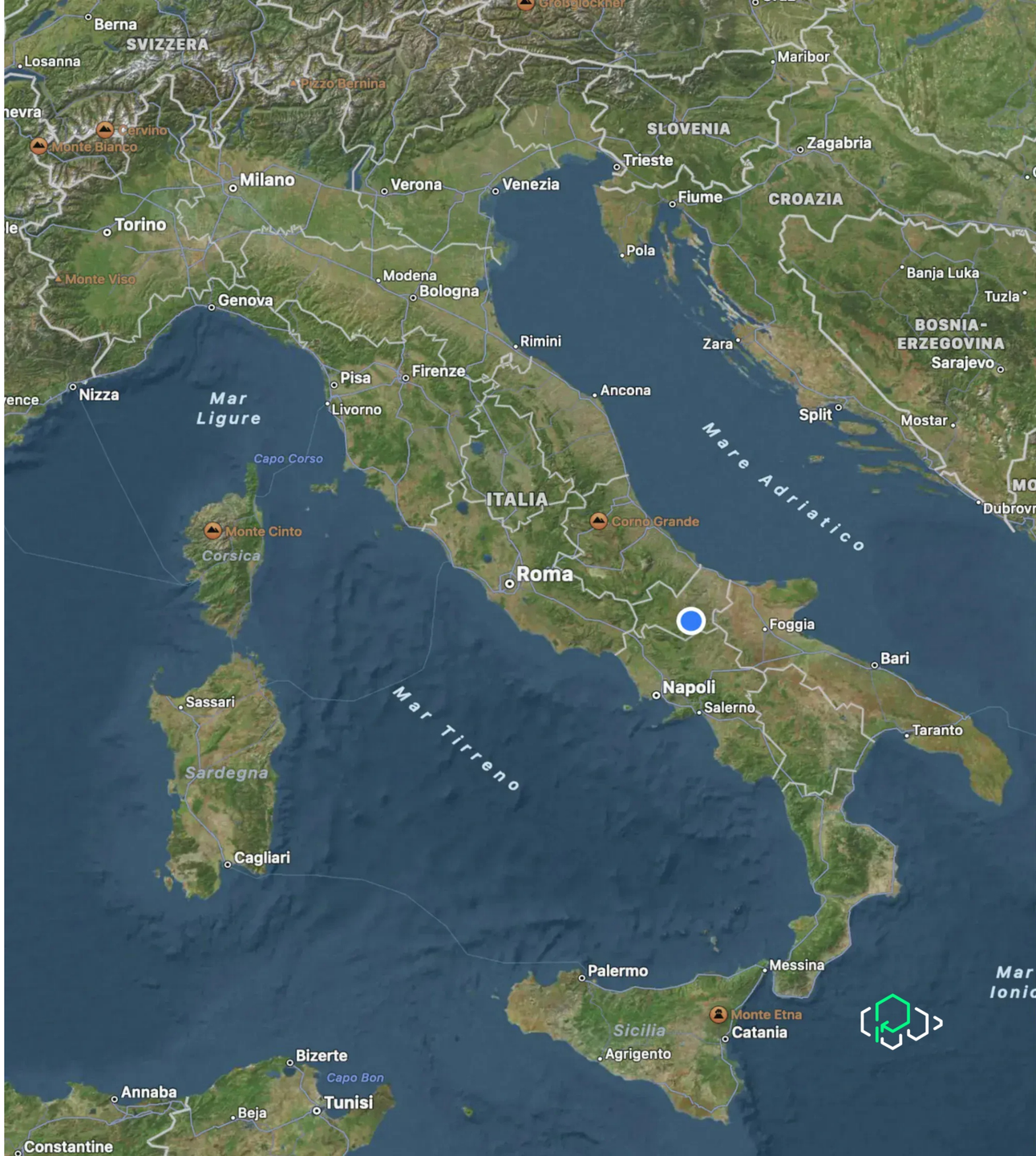
Hello, I'm **Paolo**!



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

Node.js is everywhere



The numbers speak for themselves



2+ billion downloads annually

Node.js is one of the most popular web development tools in the world.



Powers millions of applications

From startups to Fortune 500 companies, Node.js runs critical infrastructure.



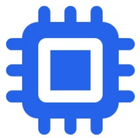
Fast and efficient

Built on V8, it delivers excellent performance for I/O-bound applications.

But there's a catch...



Single-threaded by design



One thread per process

By default, it runs JavaScript on a single event loop.



Non-blocking I/O

It's the perfect fit for web servers and APIs.



Limited for CPU-intensive tasks

Heavy computation can block the entire application.

Is this still true?





Did you hide in a cave?

Worker Threads have existed since 2018



Introduced in Node.js 10.5.0

The Worker Threads API was added in June 2018 and has been stable since Node.js 12.



True parallelism

Run JavaScript in multiple threads, each with its own V8 instance and event loop.

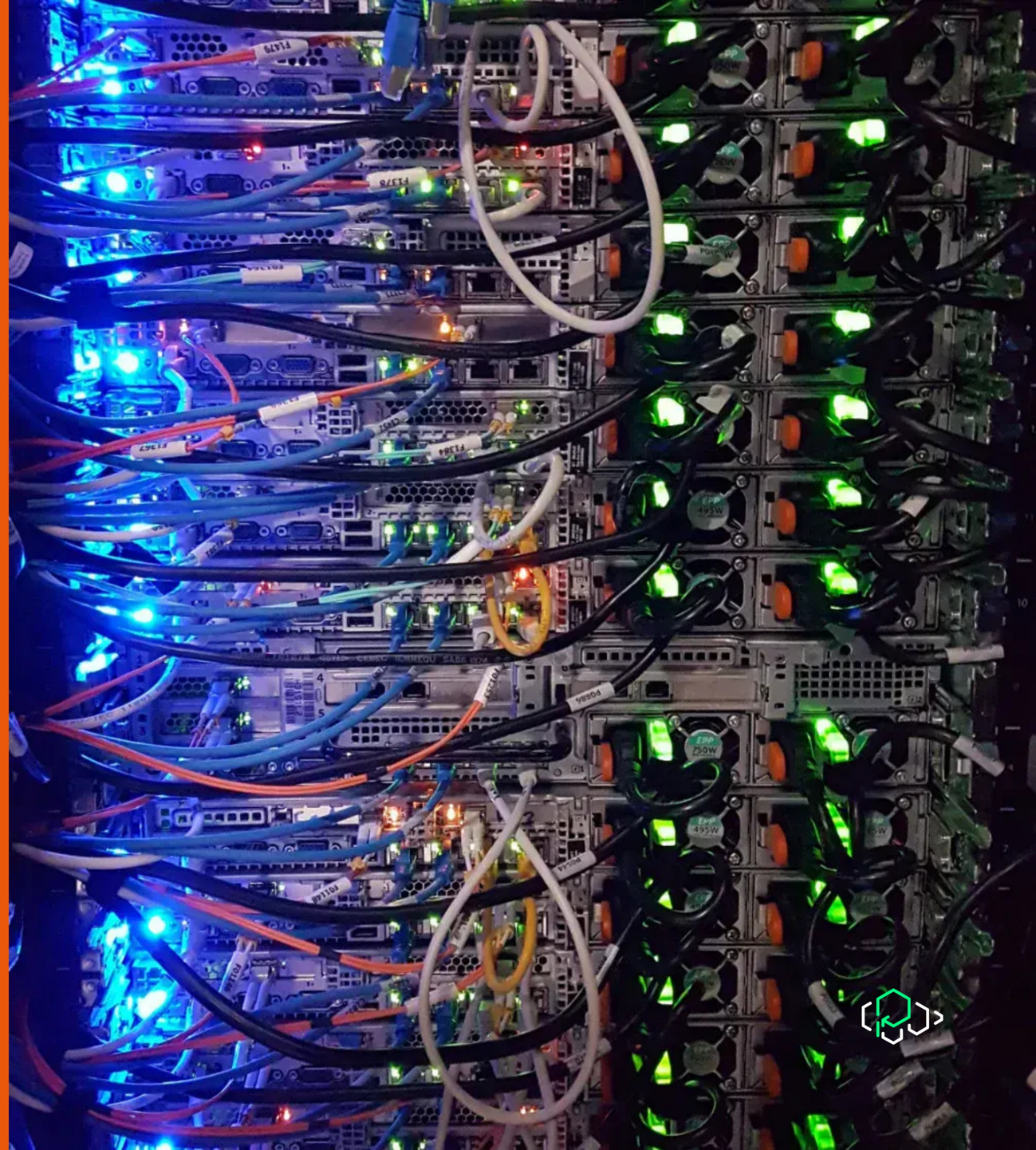


Not widely adopted

Way too many applications still don't leverage them, missing out on multi-core performance.



Running Node.js in production



The three pillars



Monitoring
Track application health and detect issues before they become critical.



Metrics
Collect and expose performance data for observability platforms.

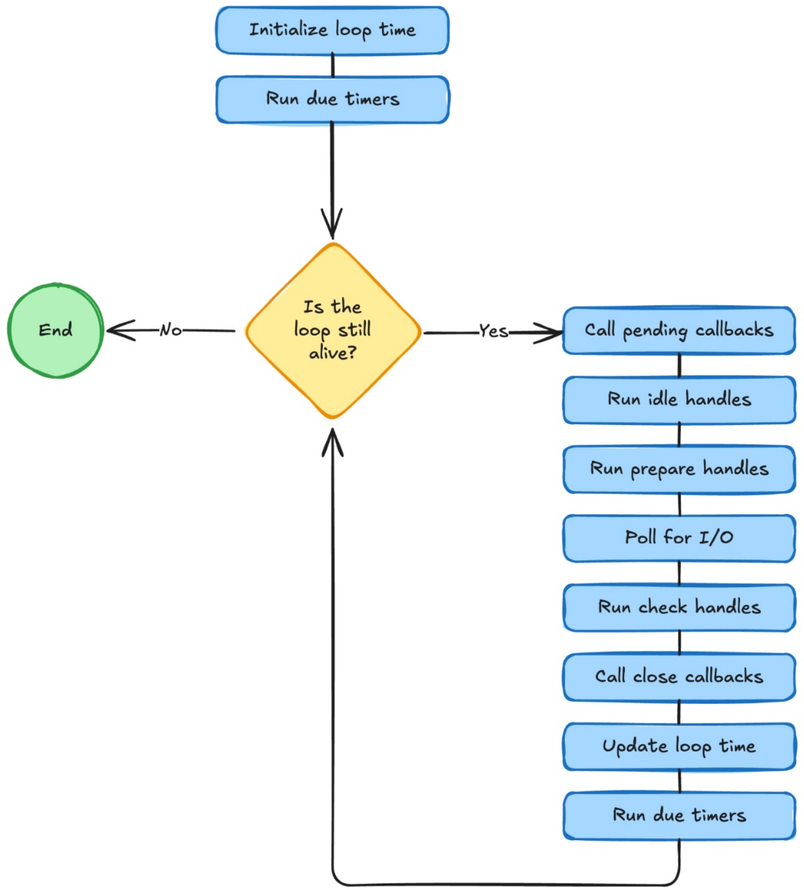


Fault tolerance
Handle failures gracefully and recover quickly from errors.

How do we monitor health?



The Node.js event loop



The event loop observation problem



The paradox

A busy event loop
cannot observe itself.



Monitoring gets blocked too

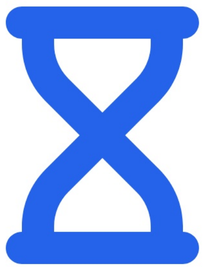
If your application is overloaded,
the monitoring code is also blocked.

Backpressure management loses effectiveness



Resource waste is "suboptimal"

You must set low thresholds
to allow monitoring to run.



Ineffective under load

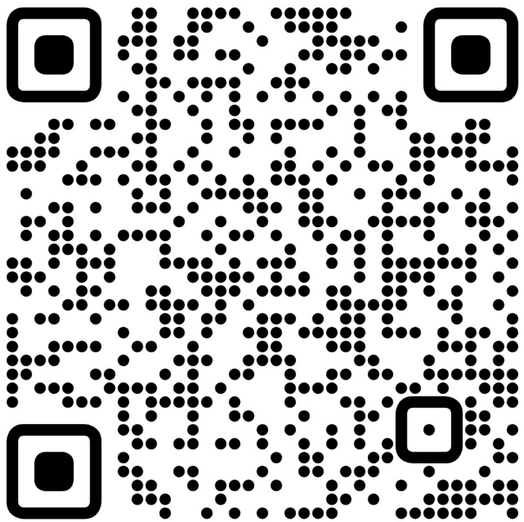
If the thread is truly stuck,
no monitoring code runs.

We need a better architecture!



Introducing Watt

A Node.js application server, done the right way!



https://www.platformatichq.com/watt

Here's our secret sauce!



Move applications to threads

Each gets its own separate worker thread.



Main thread as coordinator

The main thread manages workers and routes requests.

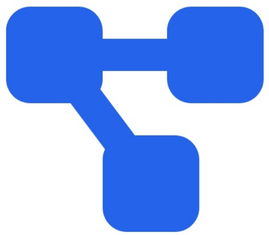


External observation

The coordinator can monitor worker health from outside.

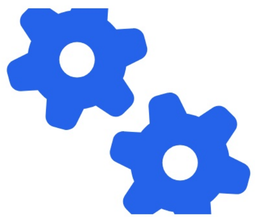


How does it work?



Main thread (coordinator)

Monitors workers
and handles their lifecycle.



Worker threads

Run applications, report health status,
and support independent restart and scaling.

What about metrics?



Prometheus server on the main thread



Dedicated metrics server

Prometheus metrics are exposed from the main thread, not from worker threads.



Avoid the paradox

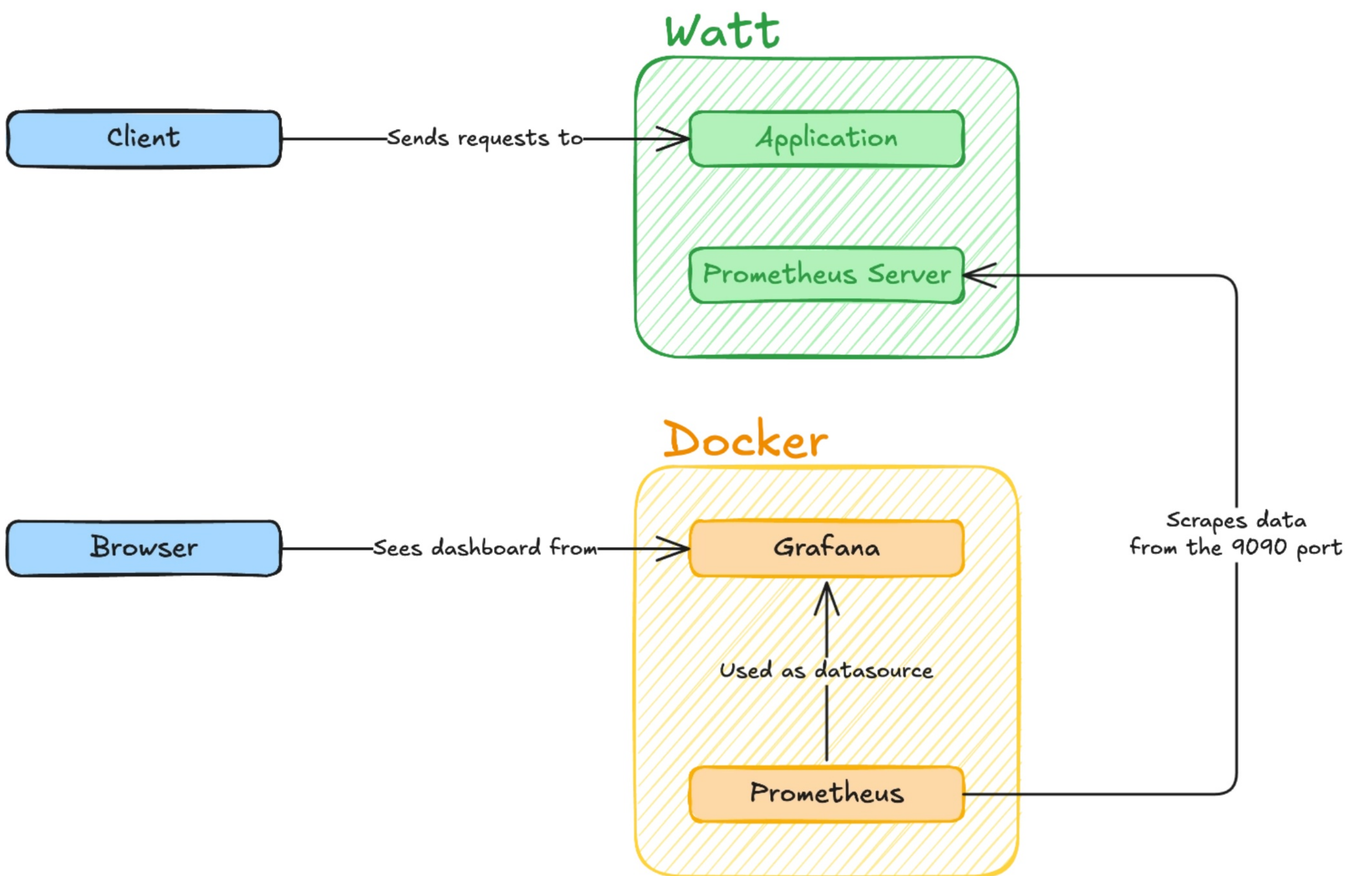
A blocked worker cannot report its own metrics reliably.



Always available

The metrics endpoint always responds, regardless of worker state.

Monitoring architecture



Kubernetes probes benefit too



Reliable readiness probes

Report as ready only when workers are actually available.



Reliable liveness probes

Always respond to liveness checks, even under heavy load.



True health visibility

Kubernetes sees the real application health, not just thread responsiveness.



Handling failures



The traditional single-threaded scenario



Degradation goes unnoticed

Health checks stop responding when the thread is busy, hiding the real problem.



Critical failures are catastrophic

Memory exhaustion, a crash, or an unhandled exception takes down the entire application.



Long recovery time

During process restart, there might be downtime or performance loss due to fewer replicas.

The innovative Watt multi-threaded approach



Let it fail

If a worker thread crashes, restart it immediately.



Proactive approach

If health metrics degrade, terminate and restart the worker.



Replace before restart

Start a new thread before stopping unhealthy ones.

Why is this approach better?





Because I'm telling you!

Don't you just trust me? 😭

Seriously, why is this approach better?



Much faster recovery

Restarting a single thread is orders of magnitude faster than restarting the entire process.



Replace, don't restart

New workers start before unhealthy ones stop, ensuring zero wait time.



No request loss

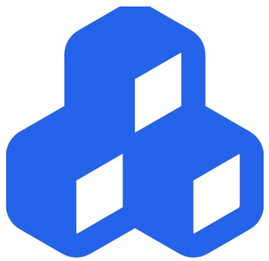
Traffic seamlessly shifts to healthy workers with no dropped connections or failed requests.





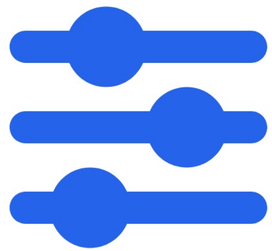
Not convinced yet?

Multiple applications, one process



Run multiple applications

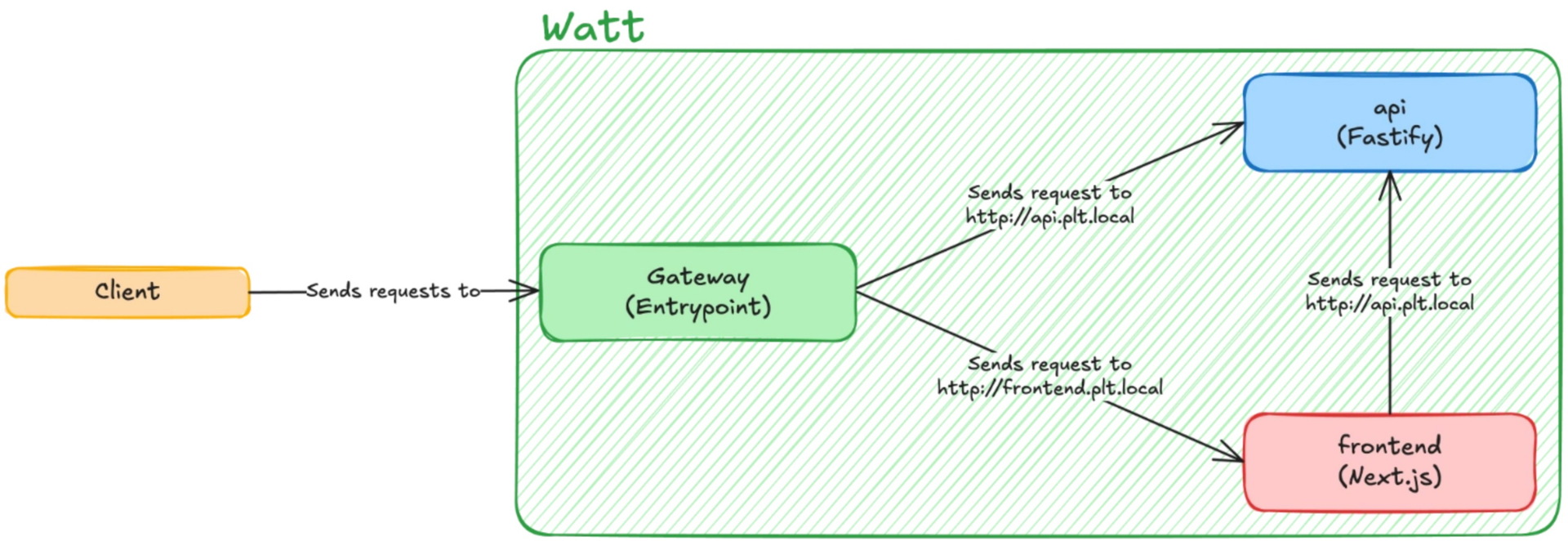
They all run in the same Node.js process, each isolated in its own worker thread.



Independent scaling

Scale each application independently based on its own load and resource requirements.

Watt in action: the mesh network



Intelligent dynamic in-process scaling



Use all available cores

Spawn workers to match available CPU cores.



Dynamic worker count

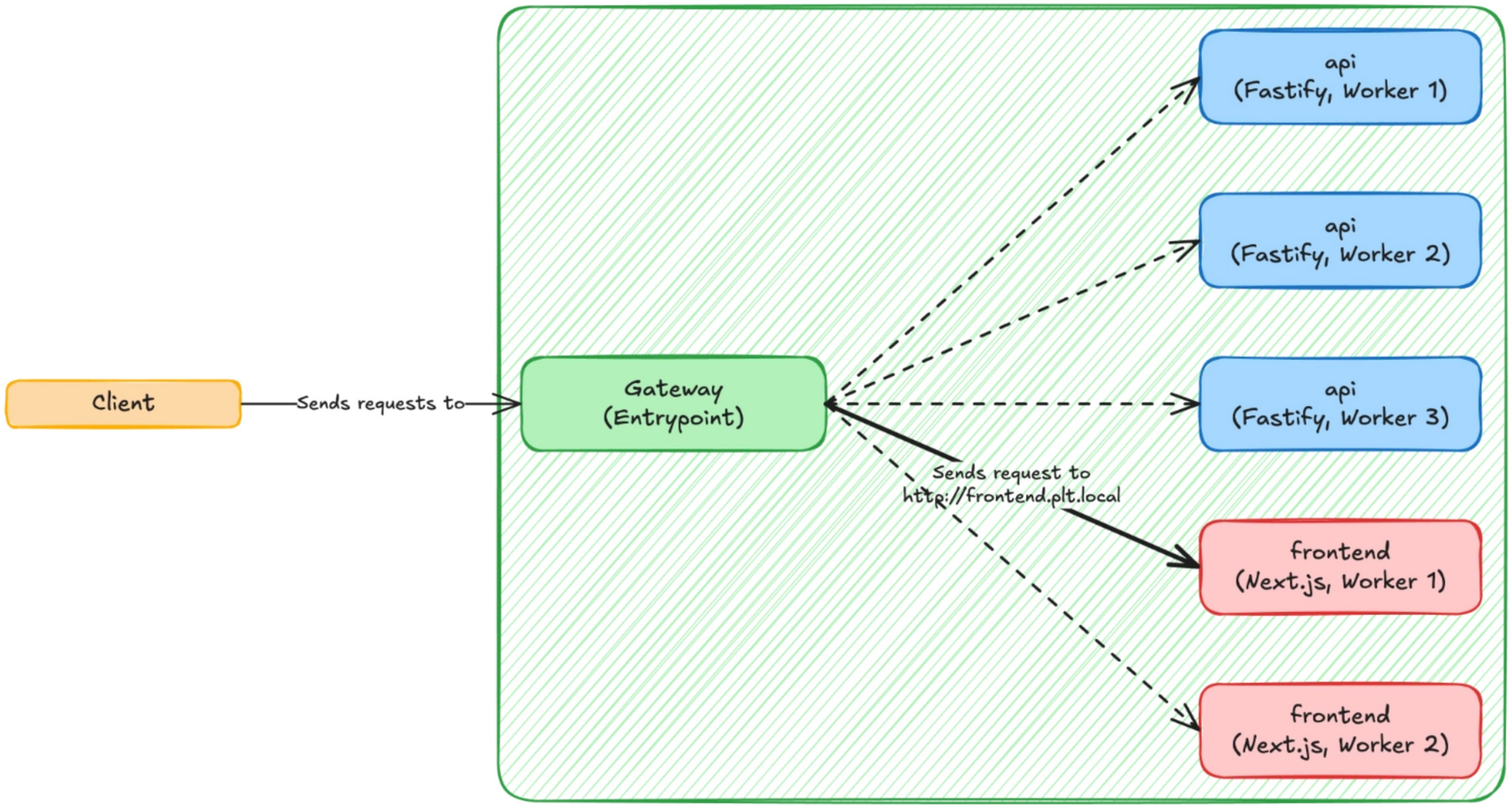
Scale workers up or down dynamically based on load and performance.



Better resource utilization

Get more from your existing infrastructure before scaling horizontally.

Watt in action: multiple workers

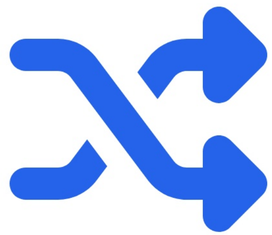


Dynamic applications



Hot-swap made easy

Add or remove applications from a running process with zero downtime.



Movable and flexible

Redistribute applications for load balancing or resource optimization without interruption.



Please, just let me go!

I promise I understood everything! 🙏

Take-home lessons



The monitoring paradox

Self-monitoring on a single thread is fundamentally flawed.



Separation of concerns

Application logic, coordination, and monitoring should be isolated.



Let it fail

Fast restarts are better than trying to recover unhealthy processes.

One last thing™

“Success is often achieved by those who don't know that failure is inevitable.”

Coco Chanel





Thank you!

Paolo Insogna

Node.js TSC, Principal Engineer

@p_insogna

paolo.insogna@platformatic.dev

Platformatic