Platformatic

View online

Download PDF

Achieving 93% Faster Next.js in (your) Kubernetes with Watt

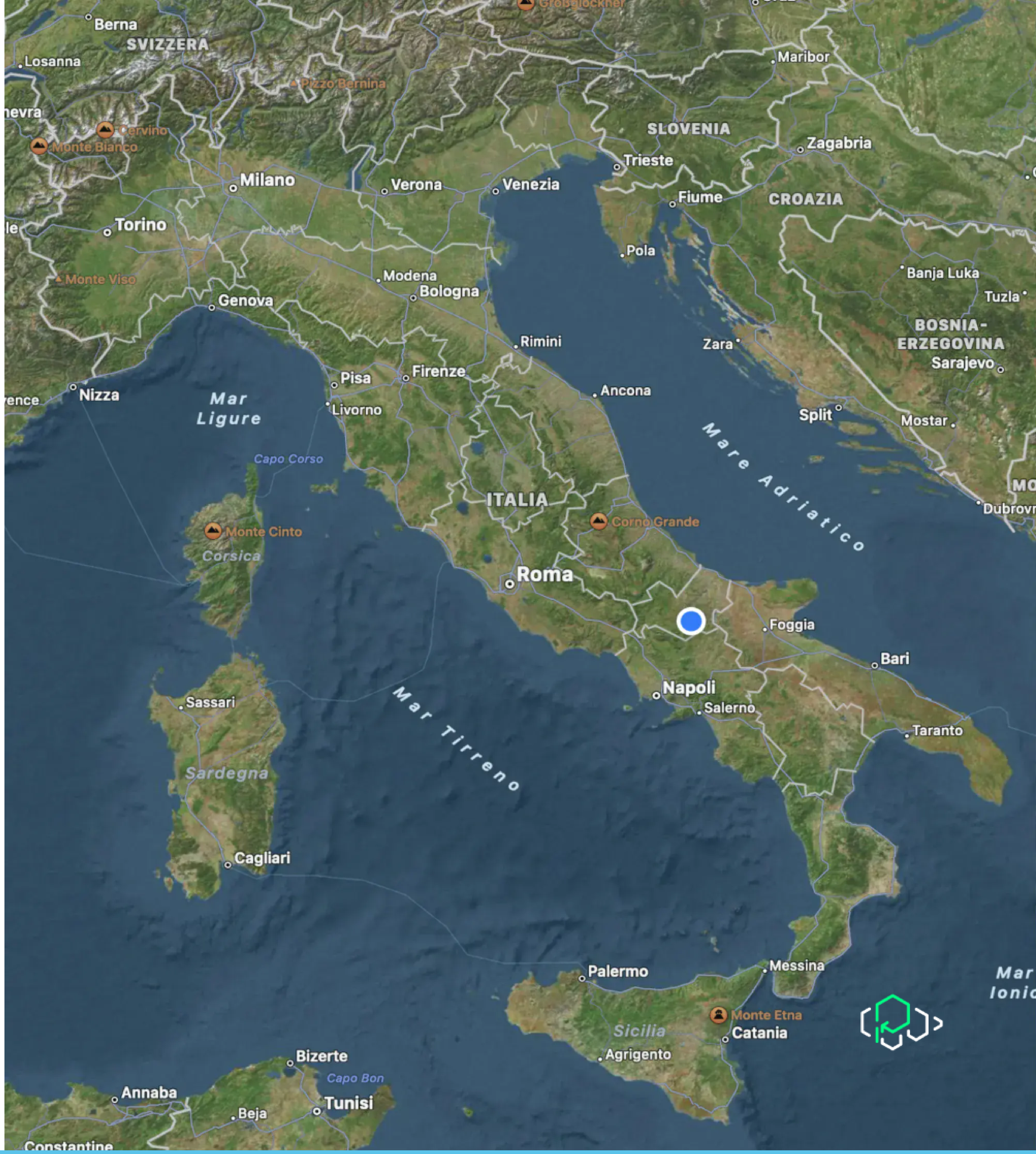
Paolo Insogna
Node.js TSC, Principal Engineer

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



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

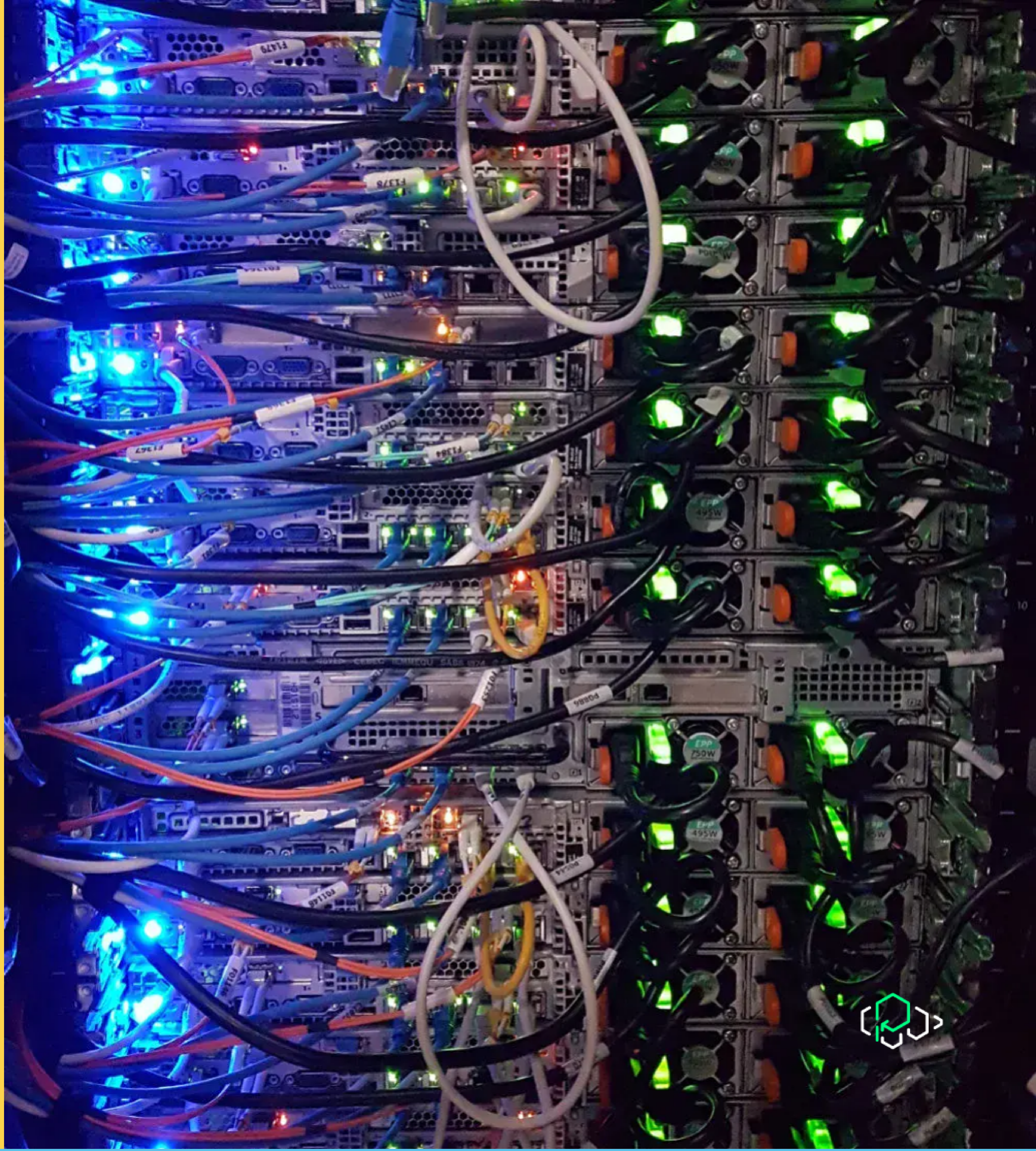
This is supported from Node.js 10.5.0 (June 2018).

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 offload CPU-intensive tasks out of the main thread.

Why do we care?

How do you scale Node.js in production?



A familiar story at scale

1. **Traffic spikes create uneven load**
   Some pods are at 100% CPU while others are at 30%. Error rate climbs to 8%.

2. **Over-provisioning costs money**
   You add 50% more pods to handle spikes. Cloud bill grows but the problem is not solved.

3. **This is a revenue problem**
   Latency across API calls leads to abandoned carts and churned customers.

The Cluster Module: How It Works (1/2)

Introduced in Node.js 0.6 (2011) to scale across multiple CPU cores.

Master process coordination

It distributes connections with a round-robin policy.

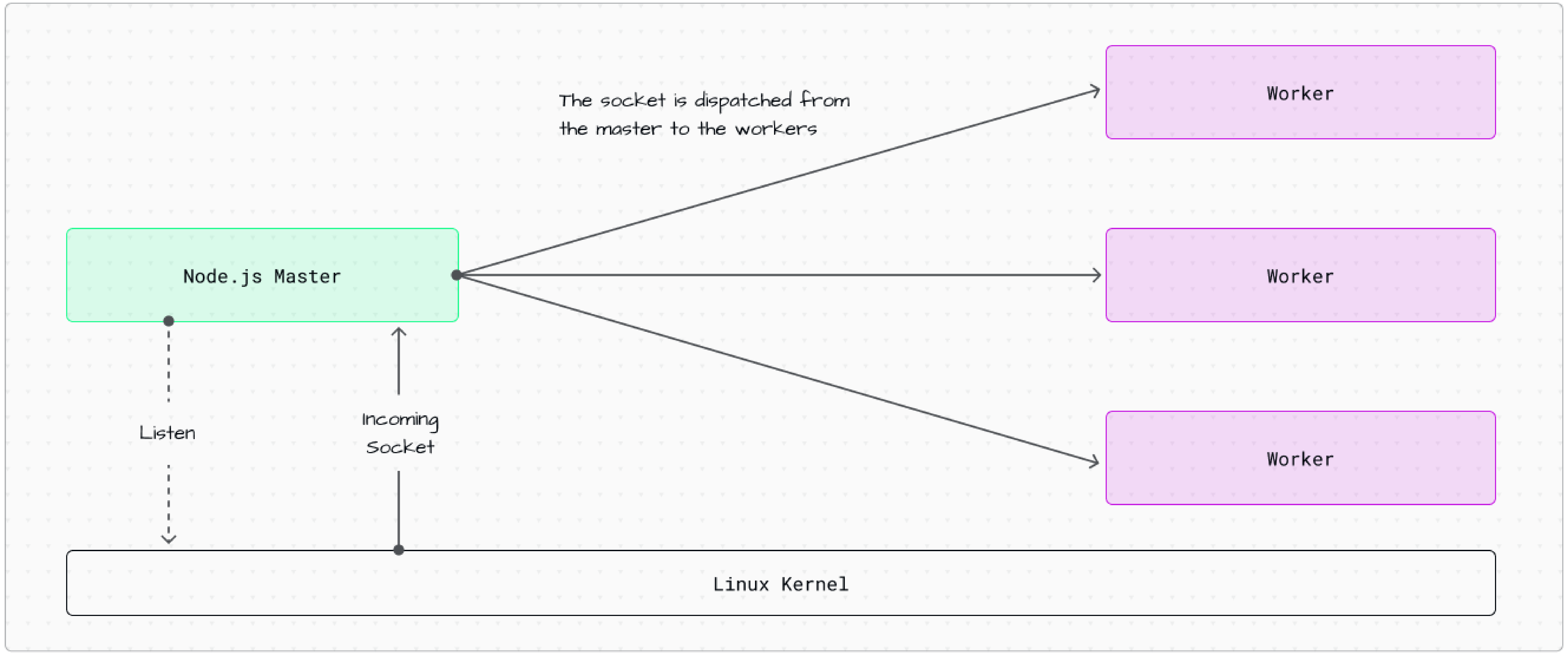
IPC adds ~30% overhead

Every connection is transferred to workers via Unix domain sockets.

PM2 makes it easier

Built on top of the cluster module. It inherits the same overhead.

The Cluster Module: How It Works (2/2)



The Early Rejection Problem

Requests enter the event loop queue

After TCP accept, the request waits for its turn to be processed.

Cannot reject until processing begins

Once in the queue, requests consume resources, which affects everyone during overload.

Ideal servers reject early with 503

Load balancers can then route traffic elsewhere. In Node.js this is difficult.

Why Next.js Makes This Worse

Request context required first

SSR needs headers, cookies, and query params before making decisions.

Dynamic route matching happens after accept

Next.js middleware runs after request acceptance. Cannot reject before knowing what to do.

Data fetching dependencies

Server components require the request to be in-flight.

The Compounding Effect

These problems multiply when combined with traditional scaling approaches.

With cluster/PM2

Every request pays the ~30% IPC overhead, even when not overloaded.

With single-CPU pods

Isolated queues compound load imbalances. No work sharing.

Latency compounds across API calls

Three sequential calls at 180ms each = 540ms wait. Churned customers in SaaS.

We solved this.

The Technical Foundation - SO_REUSEPORT

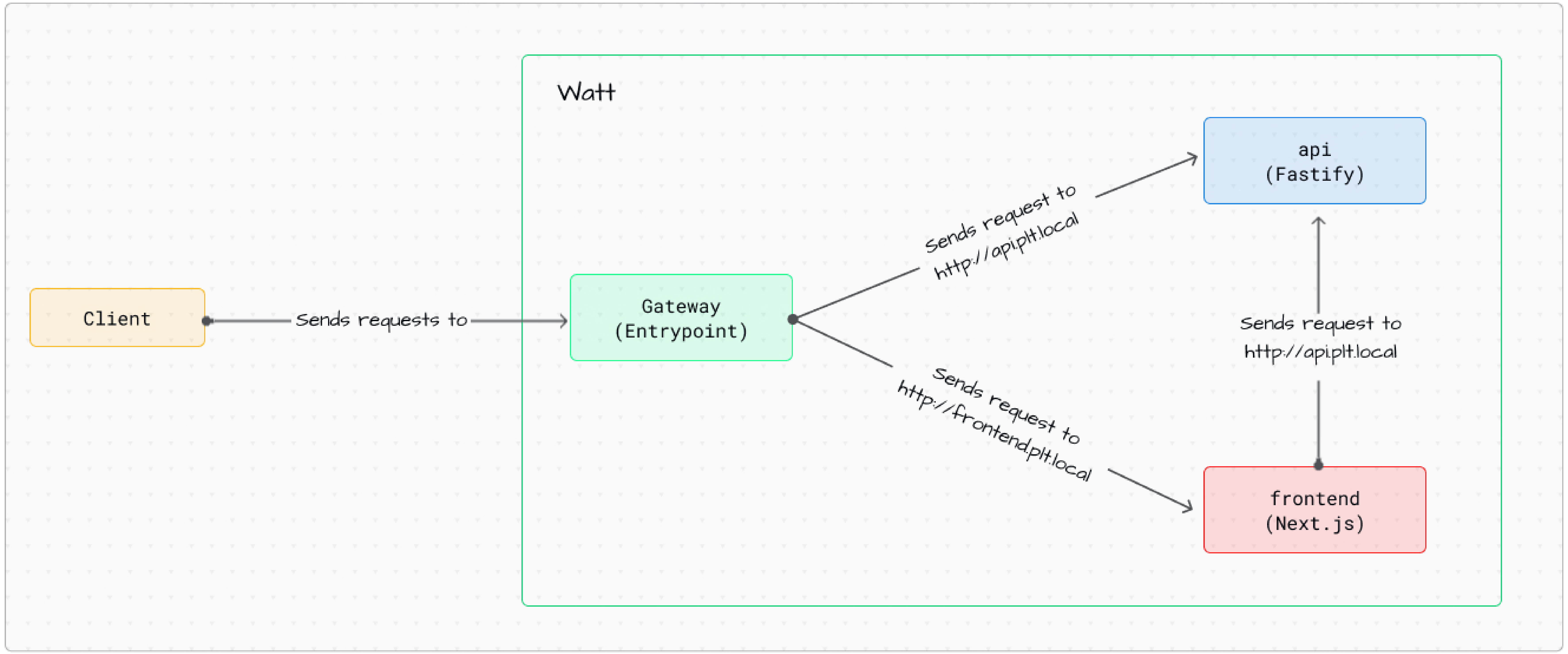
Kernel-level hash-based distribution
Calculates a hash from source IP, port, destination IP, and port to select a worker.

Connection affinity and even distribution
The same client always reaches the same worker. Zero coordination needed.

One flag enables it all
Set `reusePort: true` on the HTTP server and the kernel handles the rest.

Available since Linux kernel 3.9 (April 2013)
It fundamentally changes how connections are distributed.

Introducing Watt, the Node.js application server



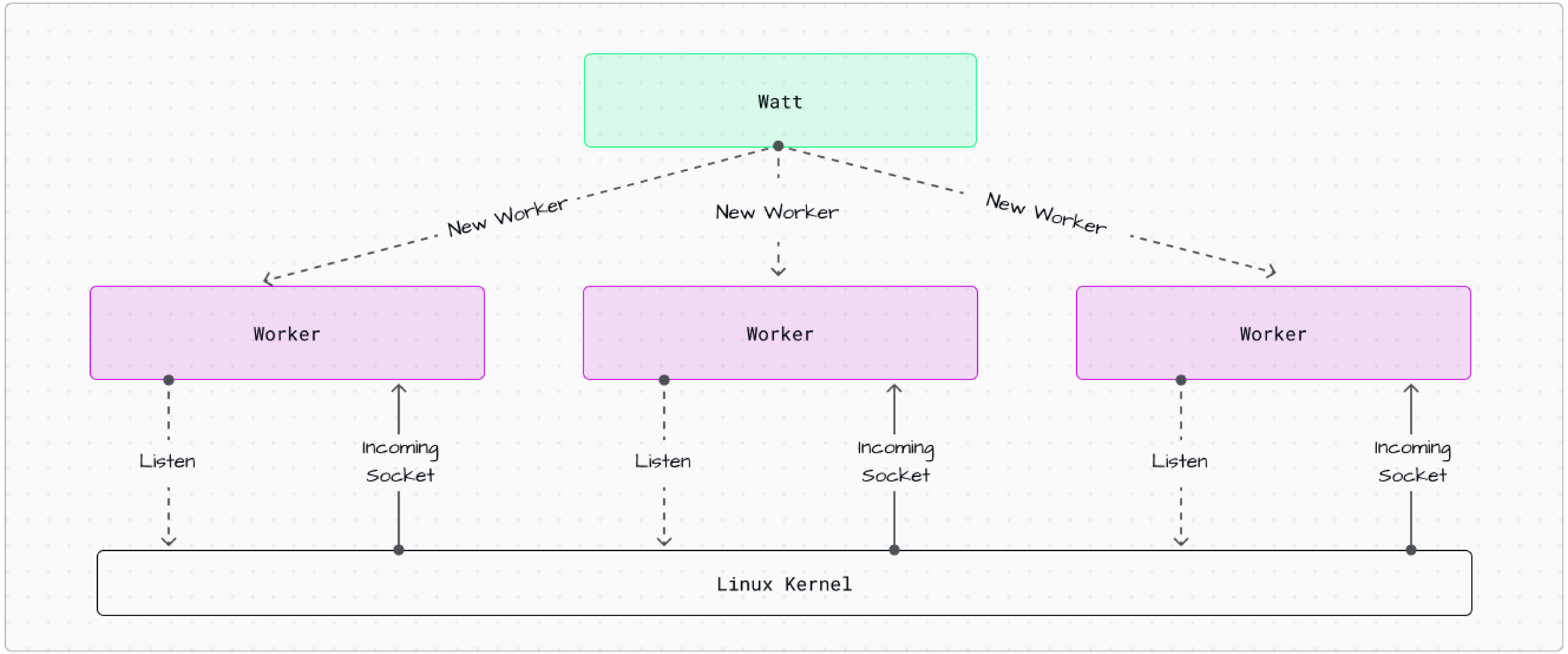
What is Watt?

Leverages SO_REUSEPORT
Workers accept connections directly. No more IPC overhead.

Multi-service architecture
Run multiple Node.js services with inter-thread communication.

Production-ready
Built-in monitoring, logging, tracing, and health checks.

Watt: Architecture



Process Orchestration

Automatic restart of crashed processes

Worker failures are detected and restarted automatically without manual intervention.

Graceful shutdown handling

Coordinated shutdown ensures requests complete before workers terminate.

Health monitoring and metrics

Built-in monitoring tracks worker health and performance metrics in real-time.

Watt: Automatic Health Restarts

Event loop failure detection
Monitors event loop health.
Detects unresponsive workers.

Heap exhaustion handling
Detects memory issues before
they crash the entire pod.

Zero downtime restart
Failed workers are replaced in-
place. Others are unaffected.

Watt: Shared HTTP Cache

Cross-worker cache sharing
All workers access the same cache. No duplicate cache entries across processes.

Reduces redundant requests
Cached responses are reused across all workers, minimizing backend load.

Improves response times
Cache hits serve responses instantly without processing overhead.

Deploying Watt in Kubernetes

Two-Layer Architecture

Layer 1: Kubernetes Service

Distributes new TCP connections across pods using a round-robin or configured algorithm.

Layer 2: SO_REUSEPORT within each pod

The kernel distributes connections across workers in a pod via hash-based selection.

Better statistical multiplexing than isolated single-CPU pods

The approach provides better load distribution and resource utilization.

Independent Event Loops

Independent processing
A worker can handle slow requests without affecting others.

Less variance
Request processing time variance is isolated to individual workers.

Better utilization
No single request can block the entire pod anymore.

Resource Sharing Within Pods

Kernel page cache
File system operations benefit from shared page cache. Less disk I/O across workers.

Memory for binary code
Application code is loaded once in memory. Lower memory footprint per worker.

Single network namespace
Lower context switching overhead. Network operations share the same kernel structures.

What about performance?

Benchmark: Summary

Next.js Application

Real Next.js application on AWS EKS. Sustained load of 1,000 req/s for 120 seconds.

AWS EKS Cluster

3 nodes: m5.2xlarge instances (8 vCPUs, 32GB RAM each)

Load Testing with k6

c7gn.large instance, constant-arrival-rate executor, 1,000 pre-allocated VUs

Benchmark: Configurations

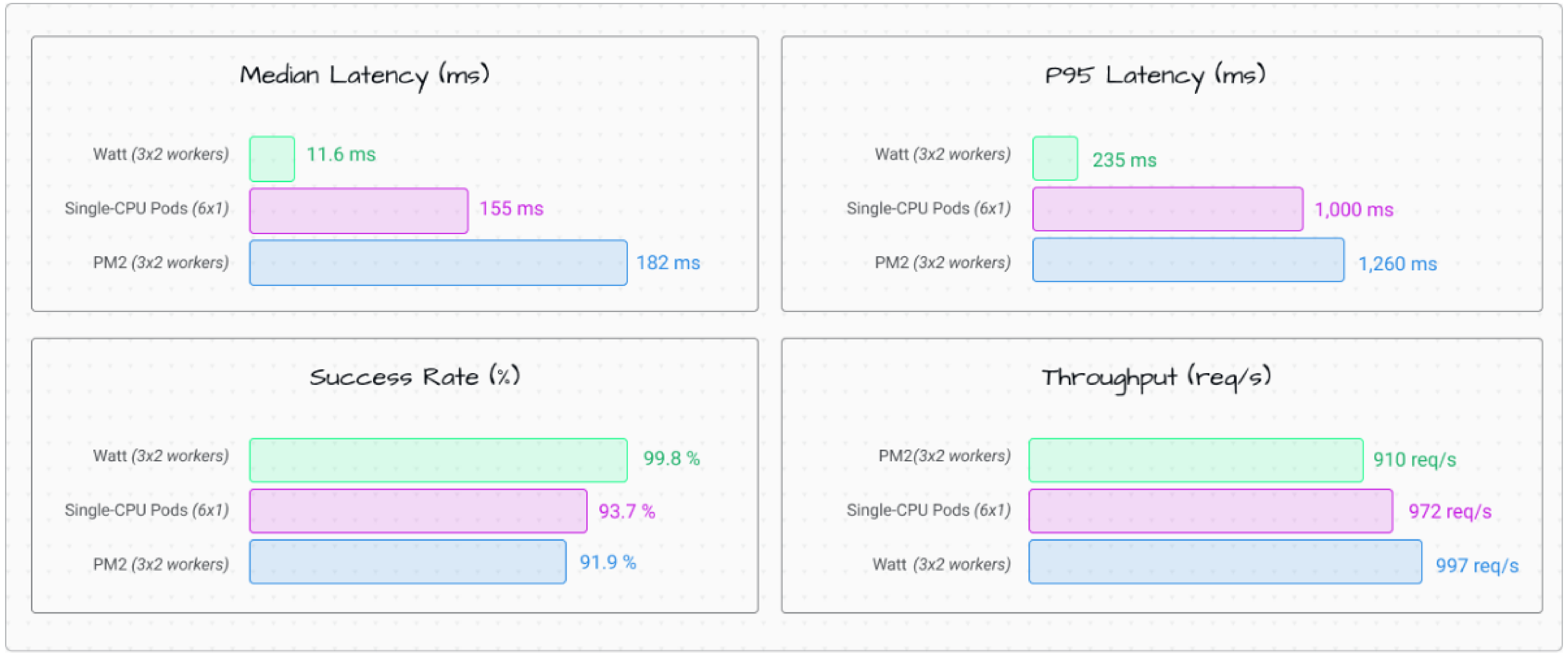
Single-CPU pods (Traditional horizontal scaling)
6 replicas x 1000m CPU = 6 total CPUs

PM2 multi-worker pods (Cluster module approach)
3 replicas x 2000m CPU with 2 PM2 workers = 6 total CPUs

Watt multi-worker pods (SO_REUSEPORT approach)
3 replicas x 2000m CPU with 2 Watt workers = 6 total CPUs

Benchmark: Results

Watt dramatically outperforms both traditional approaches.



Latency Performance

93.6% faster median latency compared to PM2

From nearly 200ms to sub-15ms response times.

81.3% faster P95 latency compared to PM2

Consistent performance even at higher percentiles.

92.5% faster than single-CPU pods

Best of both worlds: scaling and performance.

Throughput and Reliability

9.6% more throughput than PM2

Same CPU resources, much better utilization.

99.8% success rate

Near-perfect reliability under sustained load.

How is that possible?

Why PM2 Underperforms

Master process acts as internal load balancer via IPC

Every request transfers via Unix domain sockets.

About 30% overhead on every request

This overhead applies universally, even when the server is not overloaded.

Why Single-CPU Pods Underperform

Isolated queues compound imbalances

Each pod operates independently. Round-robin creates uneven distribution.

No work sharing between pods

One pod drowns at 100% CPU while another idles at 30%. Cannot rebalance.

Watt's Advantages

Zero-overhead kernel distribution

SO_REUSEPORT eliminates IPC coordination. Workers accept connections directly.

Shared accept queue with statistical multiplexing

Workers within each pod share work naturally. Better distribution than isolated pods.

99.8% reliability under load

Architecture handles burst traffic effectively. Near-perfect success rate at higher loads.

Getting Started with Watt in Kubernetes

Ready to achieve these performance gains in your own applications?
Follow these steps to deploy Next.js in Kubernetes with Watt.

https://docs.platformatic.dev/docs/guides/deployment/nextjs-in-k8s

One last thing™

"You are always a student, never a master. You have to keep moving forward."

Conrad Hall

Thank you!

Paolo Insogna

Node.js TSC, Principal Engineer

@p_insogna

paolo.insogna@platformatic.dev

Platformatic