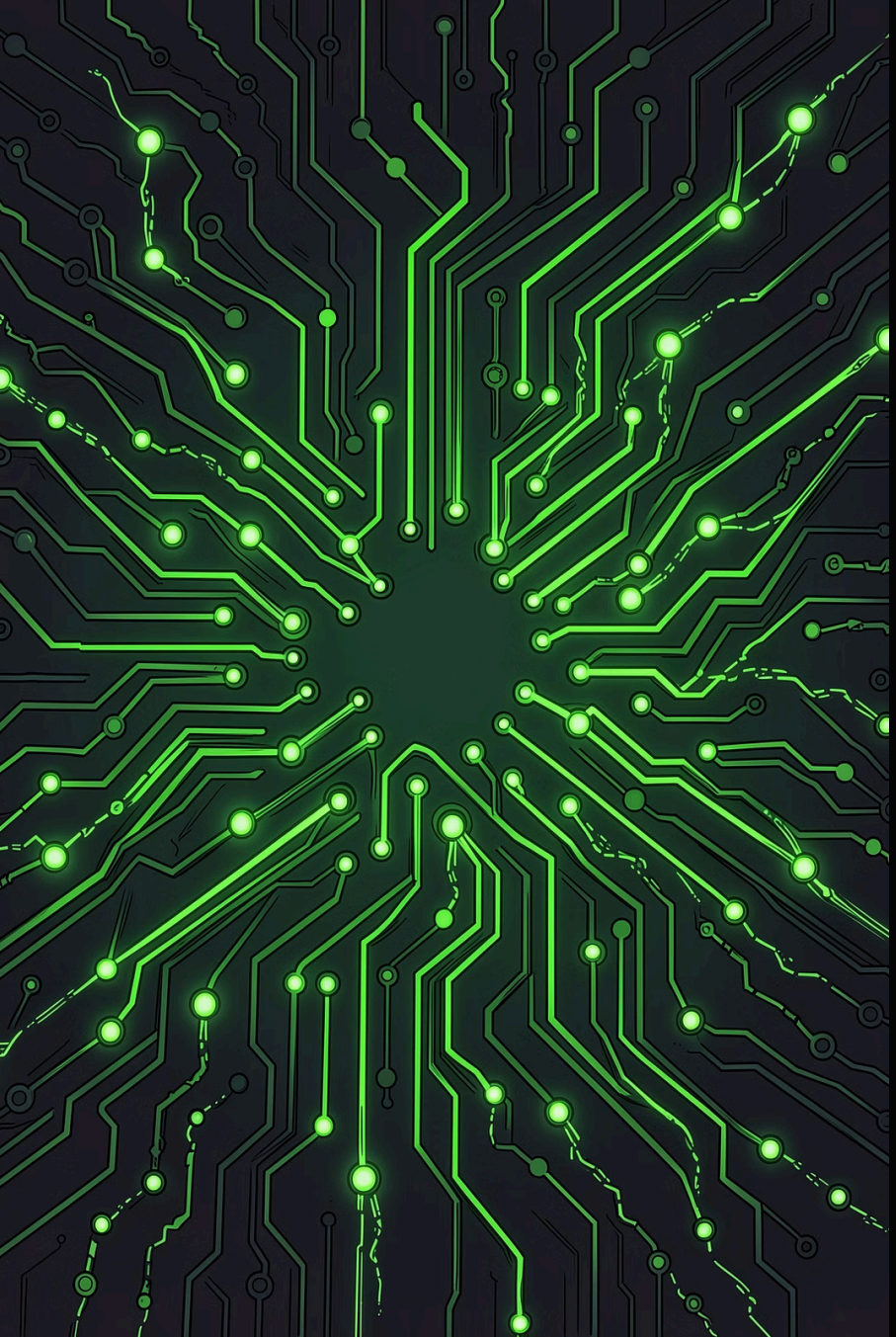


How Node.js Works Behind the Scenes

Node.js Architecture: Inside the Event Loop

A deep dive into the runtime, event loop, async I/O, and the mechanics that make Node.js fast and non-blocking.

ARCHITECTURE REFRESHER

NODE.JS INTERNALS

What We'll Cover

Eight key concepts — from the runtime stack to async I/O.

01

Node.js Runtime

V8 Engine + Node APIs + libuv

02

Call Stack

Single-threaded execution model

03

Web APIs / Node APIs

Delegating slow tasks

04

libuv + Thread Pool

Background async operations

05

Callback Queue

Task Queue for async callbacks

06

Microtask Queue

Promise callbacks — always first

07

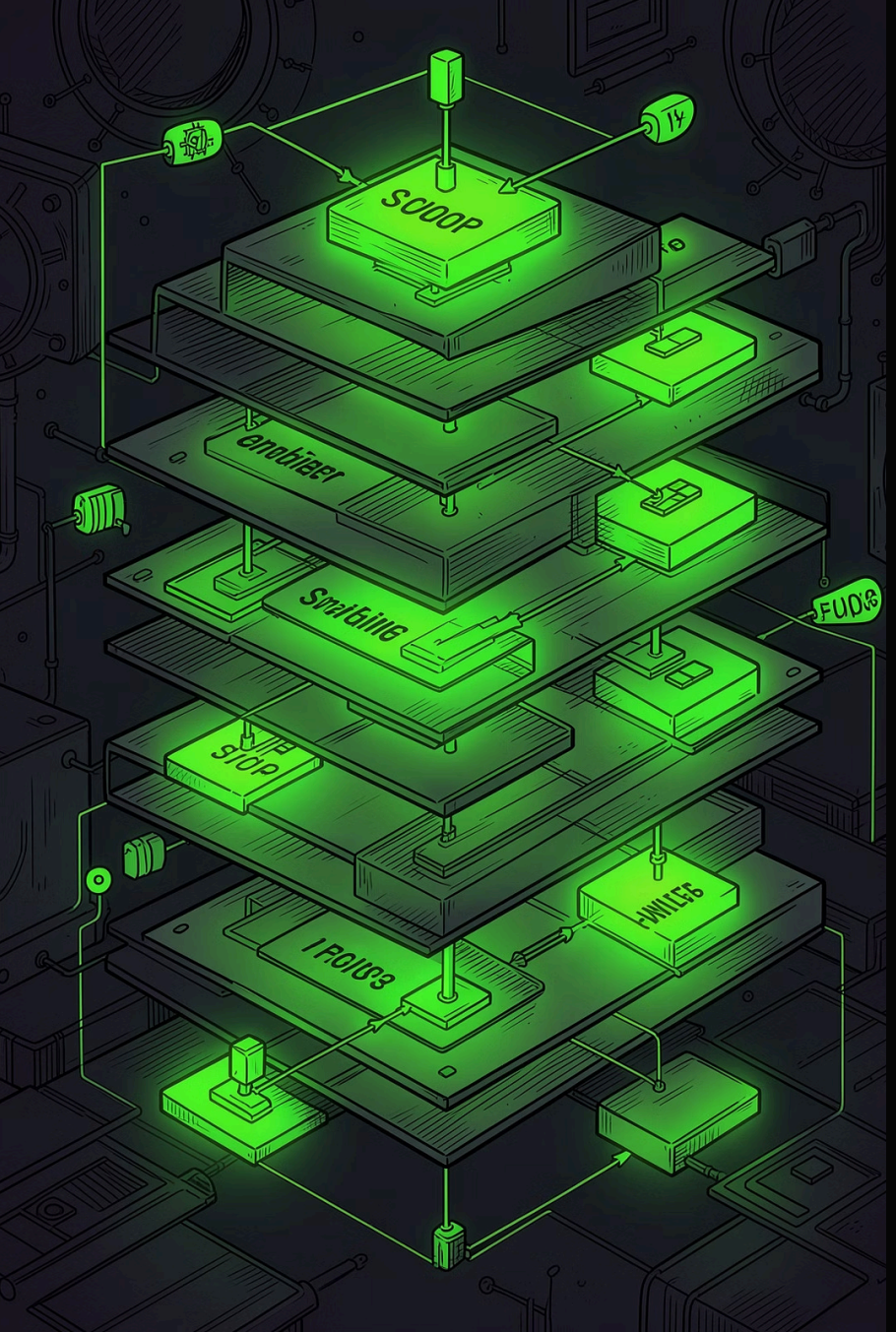
Event Loop

Orchestrating the flow

08

Blocking vs Non-Blocking I/O

Why Node.js doesn't freeze



The Node.js Runtime

Three components working together — not just V8.

V8 Engine

Google's JavaScript engine. Compiles JS to native machine code and executes it at high speed.

Node APIs

Built-in modules — `fs`, `http`, `net` — that expose OS-level capabilities to JavaScript.

libuv

C library providing the event loop, async I/O, and a thread pool for non-blocking operations.

The Call Stack

How It Works

The call stack is a **LIFO** structure — Last In, First Out. JavaScript is single-threaded, so only one function executes at a time. If a function calls another, it's pushed on top. When it returns, it's popped off.

- **Push:** Function is called → added to the top
- **Pop:** Function returns → removed from the top
- **Stack Overflow:** Infinite recursion fills the stack

Long-running code blocks the stack — and the entire thread. That's why async offloading matters.

Web APIs / Node APIs

Delegating Slow Tasks

When JavaScript encounters a slow operation — a timer, a file read, or a network request — it **does not wait**. It hands the task off to Web APIs (in browsers) or Node APIs (in Node.js), which run outside the main thread.

Once complete, the API registers a callback to be executed later — keeping the Call Stack free.

setTimeout / setInterval

Scheduled timer callbacks

fs.readFile / fs.writeFile

Asynchronous file system I/O

http.get / fetch

Network and HTTP requests

DOM / EventTarget

Browser event listeners

libuv + Thread Pool

libuv is the C library powering Node.js's async I/O. It manages the event loop and maintains a **thread pool** (default: 4 threads) for expensive operations.

1

JS Thread

Hands off async task to libuv

2

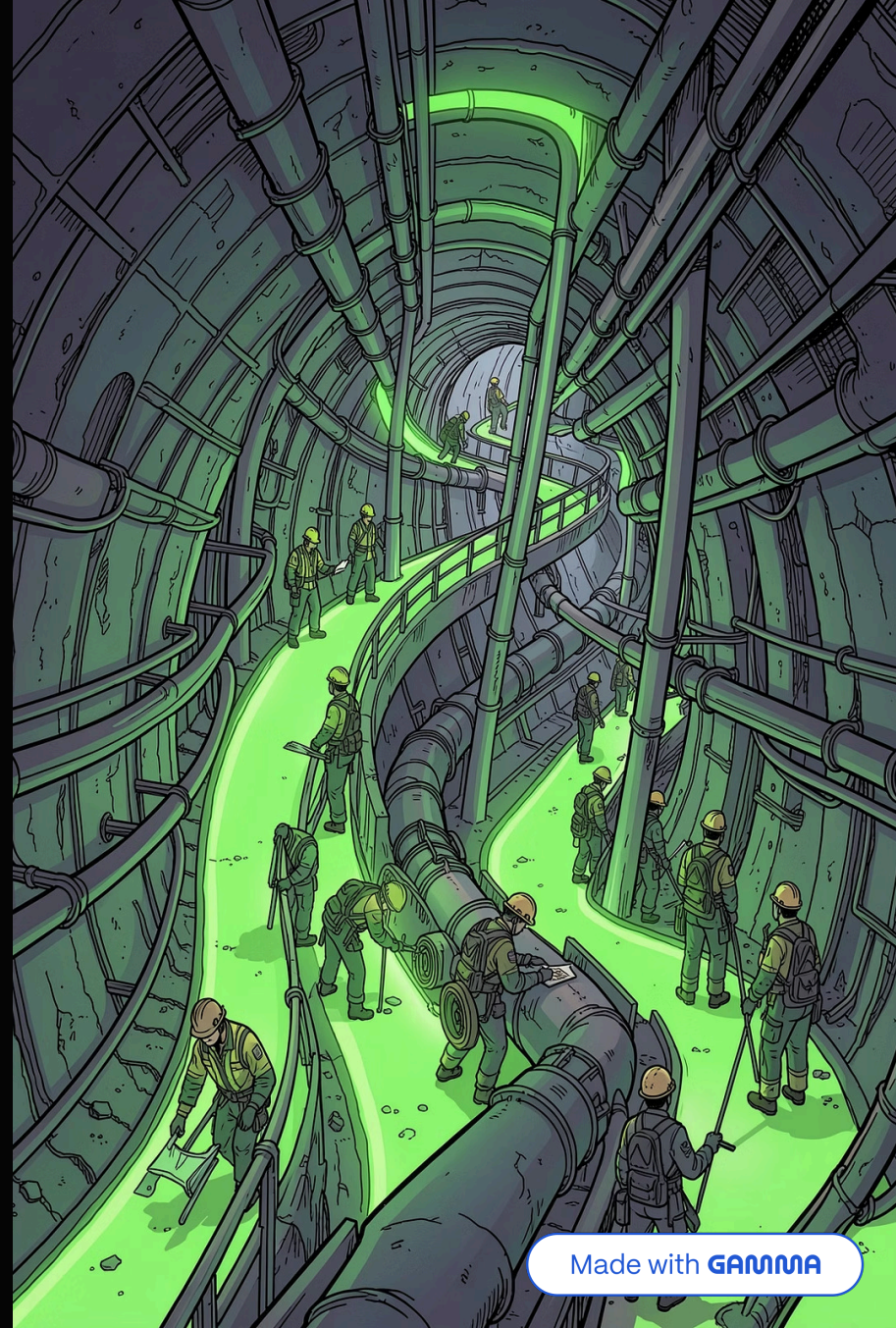
Thread Pool

Worker threads execute I/O in background

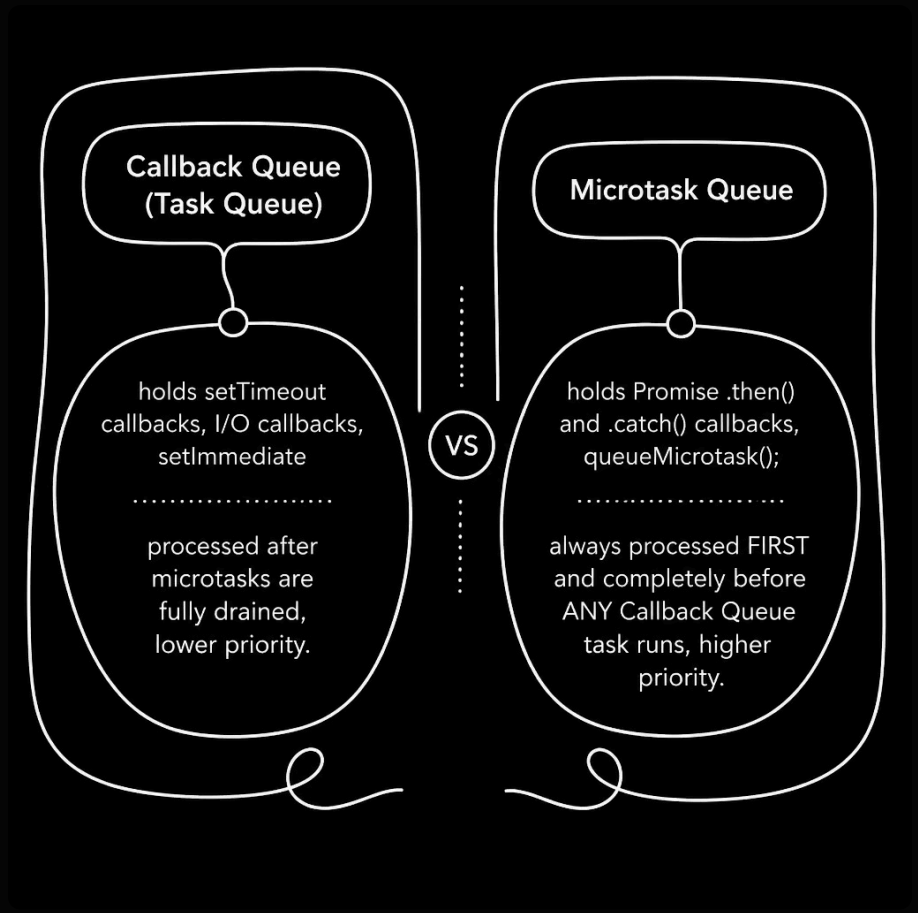
3

Completion

Result queued for callback execution



Callback Queue vs Microtask Queue



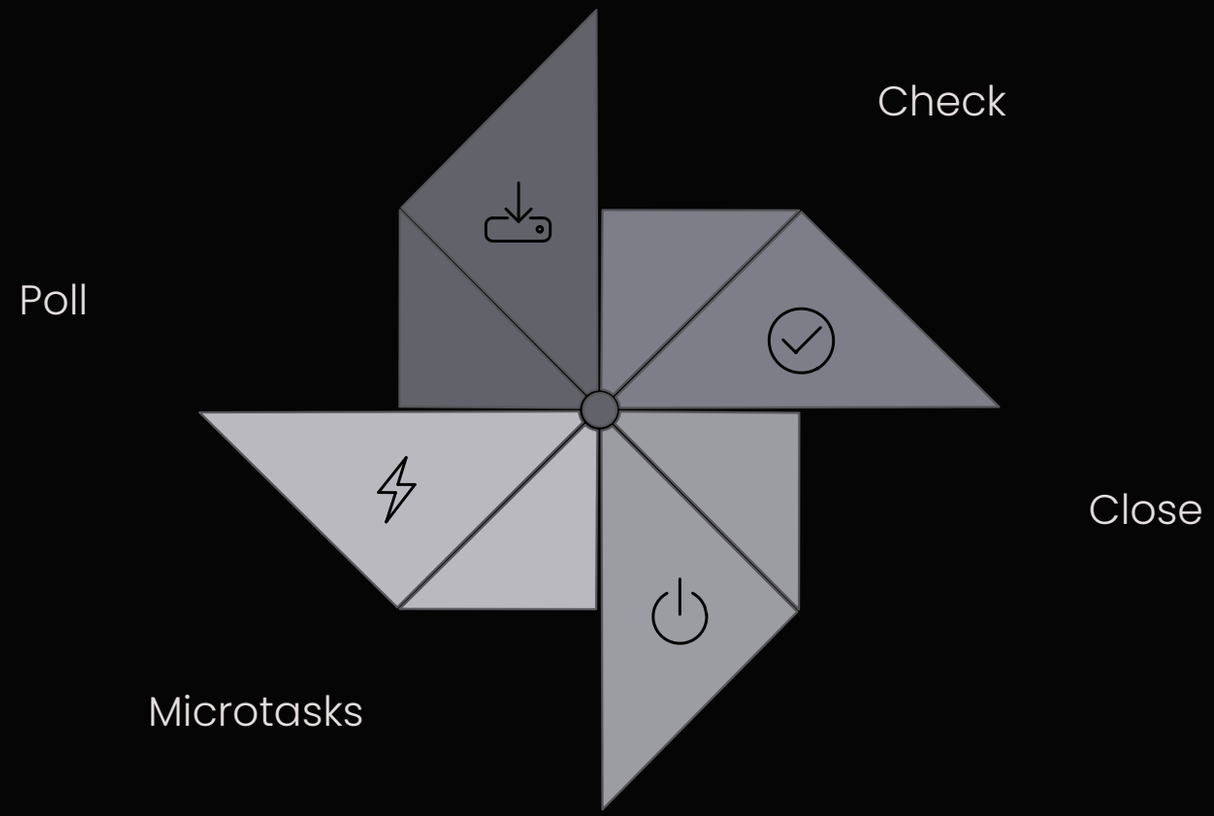
Priority Is Everything

The **Microtask Queue** always drains *completely* before the Event Loop touches the Callback Queue. This is why `Promise.then()` callbacks fire before `setTimeout(fn, 0)` — even when the timeout is zero.

- **Microtasks:** Promise callbacks, `queueMicrotask()`, MutationObserver
- **Macrotasks:** `setTimeout`, `setInterval`, I/O callbacks, `setImmediate`

The Event Loop

The Event Loop is the heartbeat of Node.js — a continuous loop that coordinates async work.



The loop checks if the Call Stack is empty, drains the Microtask Queue first, then processes the next phase's callbacks. It never stops — even when idle.

Blocking vs Non-Blocking I/O

Non-Blocking I/O — Node.js Default

Node initiates an I/O operation and immediately returns. The thread stays free to handle other requests. When I/O completes, the callback is queued.

- Server handles thousands of concurrent connections
- No thread blocked waiting on disk or network
- Uses `fs.readFile`, `http.get`, Promises

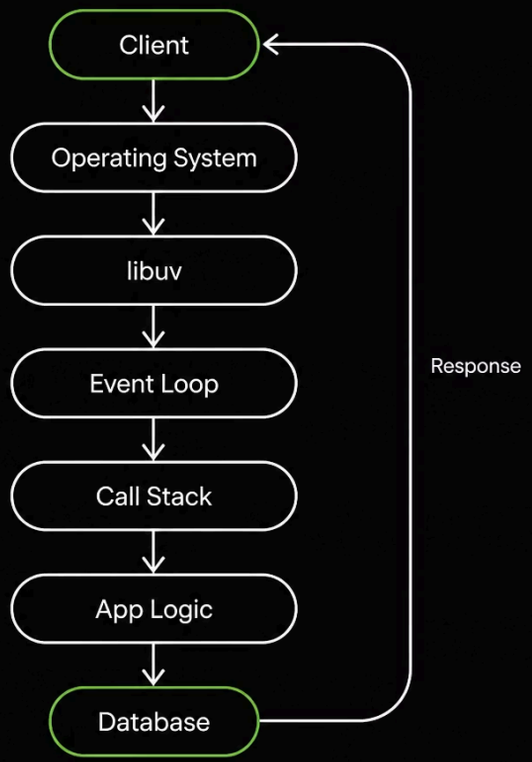
Blocking I/O — Synchronous Alternative

The thread halts until the operation completes. No other work happens during the wait — a performance killer in servers.

- Simple to reason about, dangerous at scale
- A single slow call can stall the entire server
- Uses `fs.readFileSync`, `JSON.parse` on large data

Full Architecture: Request to Response

Every request flows through the stack below — and the response loops back.



The Complete Path

A request enters through the OS, libuv handles async I/O, the Event Loop orchestrates callbacks, and the Call Stack executes your JavaScript. The response flows back — all without blocking the main thread.

- **Client → OS:** TCP connection accepted
- **libuv:** Async I/O delegated to thread pool
- **Event Loop:** Callbacks scheduled and executed
- **Call Stack:** Your JS runs synchronously
- **Database → Client:** Response returned, loop continues