

**🔧 Node.js Architecture Overview**

Node.js uses a **single-threaded, event-driven architecture** designed to build scalable network applications. It's based on:

* **Google Chrome's V8 JavaScript Engine**
* **libuv library** for I/O
* **Event Loop**
* **Thread Pool** (for expensive operations)

**🧠 Main Components of Node.js Architecture**

**1. V8 Engine**

* Developed by Google (used in Chrome)
* Converts JavaScript code to native machine code using **Just-In-Time (JIT)** compilation.
* Node.js uses V8 to run JS on the server side (outside browser).

**2. Libuv**

* A C/C++ library that provides:
  + **Event Loop**
  + **Asynchronous I/O**
  + **Thread Pool**
* It makes Node.js non-blocking by offloading expensive operations to a separate thread pool.

**3. Event Loop**

* The **core** of Node.js that handles **asynchronous** operations.
* It continuously checks the **call stack** and **callback queue** to execute tasks.

**🔄 How Node.js Handles Operations Internally**

**💥 Blocking vs Non-Blocking I/O**

| **Type** | **Example** | **Behavior** |
| --- | --- | --- |
| **Blocking** | fs.readFileSync() | Stops code execution until the task completes |
| **Non-Blocking** | fs.readFile() | Registers a callback and moves on |

Node.js encourages **non-blocking** I/O to avoid freezing the application for every I/O task.

**⚙️ Thread Pool in Node.js**

While Node.js runs in a single thread (event loop), some operations are **delegated to a pool of threads**.

**🔧 What Uses the Thread Pool?**

* File system operations (fs)
* DNS lookups (dns)
* Compression (zlib)
* Encryption (crypto)
* Others using native modules (e.g., bcrypt)

**🧵 Default Thread Pool Size**

* Node.js uses **libuv’s thread pool**, which has **4 threads** by default.
* You can increase this with the environment variable:

bash

CopyEdit

UV\_THREADPOOL\_SIZE=8 node server.js

✅ Max value: **128**

**📌 Example: Thread Pool In Action**

js

CopyEdit

const fs = require('fs');

console.time('read');

fs.readFile('./largefile.txt', () => {

console.timeEnd('read');

});

If multiple readFile operations are queued, libuv manages them in the thread pool.

**🚦 Event Loop Phases**

Node.js Event Loop runs in multiple **phases**:

1. **Timers** – Executes callbacks scheduled by setTimeout and setInterval.
2. **Pending Callbacks** – Executes I/O callbacks deferred to the next loop.
3. **Idle, Prepare**
4. **Poll** – Retrieves new I/O events; executes I/O callbacks.
5. **Check** – Executes setImmediate() callbacks.
6. **Close Callbacks** – Executes close events like socket.on('close').

**📌 Microtasks (Promises) vs Macrotasks**

js

CopyEdit

setTimeout(() => console.log('Timeout'), 0);

Promise.resolve().then(() => console.log('Promise'));

Output:

javascript

CopyEdit

Promise

Timeout

📌 Microtasks (Promises, queueMicrotask) run **after current operation but before the next event loop phase**.

**🧰 Key Node.js Internal Queues**

| **Queue** | **Example** |
| --- | --- |
| Call Stack | All executing JS code |
| Callback Queue | setTimeout, setInterval, I/O callbacks |
| Microtask Queue | Promise.then, queueMicrotask |
| Event Queue | Event listeners |

**🧪 Can We Maximize Thread Pool?**

Yes.

**✅ How to increase the thread pool size:**

bash

CopyEdit

UV\_THREADPOOL\_SIZE=16 node app.js

**⚠️ When Should You Increase It?**

* If you perform **many concurrent I/O-bound tasks** (like reading multiple files)
* **Not useful** for CPU-bound tasks — consider **Worker Threads** or offloading to separate processes

**🧱 Worker Threads (For CPU-Intensive Tasks)**

Node.js v10.5+ supports **Worker Threads** via worker\_threads module:

js

CopyEdit

const { Worker } = require('worker\_threads');

Used for:

* Image/video processing
* Hashing
* JSON parsing of massive data

**🎯 Summary of Execution Flow**

1. JS code runs in V8.
2. Expensive ops (like file I/O) go to **libuv thread pool**.
3. Once done, callbacks are sent to the **event loop**.
4. Event loop picks callbacks and runs them via V8.

**📚 Real-world Use Cases of Node.js**

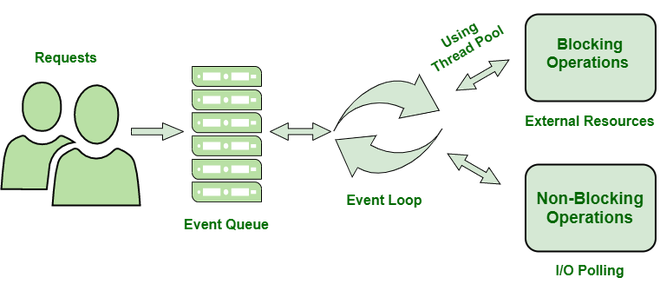
* Real-time applications (chat apps, notifications)
* RESTful APIs and microservices
* Streaming services
* Command-line tools
* IoT and sensor data systems

**📌 Node.js Is Best Suited For:**

✅ I/O-bound applications  
✅ Low-latency APIs  
✅ Real-time services  
❌ CPU-intensive operations (unless Worker Threads are used)

**🔐 Advanced Concepts**

* **Cluster Module**: To utilize multi-core CPUs
* **Child Process**: Spawn system processes
* **EventEmitter**: Custom event handling system
* **Process.nextTick()**: Schedules a callback before next event loop tick
* **Async Hooks**: Track async resources lifecycle



**1. What is Node.js?**

Node.js is a **runtime environment** that allows you to run JavaScript **outside of the browser**, typically on the server. It's built on Chrome’s V8 engine and designed for building fast, scalable, network-based applications using an **event-driven, non-blocking I/O model**.

**⚙️ 2. V8 Engine**

* Created by Google for Chrome.
* Compiles JavaScript into **high-performance machine code** using **Just-In-Time (JIT)** compilation.
* Node.js uses it to execute JS code on the server.

**🔌 3. Libuv**

* A C/C++ library used internally by Node.js.
* Provides core features like:
  + **Event Loop**
  + **Asynchronous I/O**
  + **Thread Pool**
* Enables Node.js to perform non-blocking operations (file I/O, networking, etc.)

**🔁 4. Event Loop**

* The **heart of Node.js's async model**.
* Continuously runs and checks for new tasks.
* Picks up completed tasks and executes their callbacks.
* Ensures **non-blocking, single-threaded behavior** by deferring slow tasks (like disk I/O) to the thread pool.

**🧵 5. Thread Pool**

* Background workers managed by **libuv** for handling **expensive, blocking operations**.
* Used for:
  + fs module (file read/write)
  + crypto (hashing, encryption)
  + dns (hostname lookup)
* 🧠 **Default size**: 4 threads. Can be increased using UV\_THREADPOOL\_SIZE.

**⏳ 6. Blocking vs Non-Blocking**

| **Type** | **Example** | **Behavior** |
| --- | --- | --- |
| Blocking | fs.readFileSync() | Waits until the task is done |
| Non-Blocking | fs.readFile() | Moves on immediately, uses callback |

* Node.js favors **non-blocking I/O** to avoid delays and boost performance.

**🧠 7. Microtasks vs Macrotasks**

* **Microtasks**: Promise.then(), process.nextTick()
  + Runs **immediately after current code execution**, before any timer or I/O.
* **Macrotasks**: setTimeout(), setImmediate()
  + Scheduled by the Event Loop in different phases.

**🔄 8. Phases of Event Loop**

1. **Timers** – Executes setTimeout and setInterval.
2. **Pending Callbacks** – Executes some I/O callbacks.
3. **Idle/Prepare** – Internal use.
4. **Poll** – Waits for I/O and executes them.
5. **Check** – Executes setImmediate().
6. **Close Callbacks** – Handles socket.close(), etc.

**📦 9. Node APIs**

Built-in modules that provide OS-level capabilities, like:

* fs: File system
* http: Web server
* os: OS info
* crypto: Encryption
* path: Path manipulation

**🧰 10. Worker Threads**

Used for **CPU-bound tasks** (like image processing, complex calculations).

* Allows you to run multiple JS threads.
* Avoids blocking the main thread.

**🔐 11. JWT, Cookies, Security (Middleware)**

* Node.js handles secure sessions using **JWT (JSON Web Tokens)** and **cookies**.
* Middleware (e.g., express-jwt, custom) is used to protect routes and verify user roles.

**🧪 12. Can We Increase the Thread Pool?**

Yes! Set:

bash

CopyEdit

UV\_THREADPOOL\_SIZE=8 node app.js

* Useful when you have many concurrent I/O tasks.
* Default is 4 threads; max is 128.

**🧬 13. Cluster Module**

* Enables Node.js to **scale across multiple CPU cores**.
* You can run multiple instances of the Node app in parallel.

**⚖️ 14. Child Process Module**

* Lets you run **external programs or system commands**.
* You can use spawn, exec, or fork to manage subprocesses in Node.js.

**🔄 15. Async Hooks (Advanced)**

* Track the **lifecycle of asynchronous operations** (for debugging or advanced monitoring).
* Useful in logging tools, profilers, etc.

**💡 Summary**

Node.js is **lightweight**, **efficient**, and ideal for real-time and data-intensive apps due to its **non-blocking event-driven architecture**. Understanding the **V8 engine, event loop, libuv, and thread pool** is key to mastering Node.js.