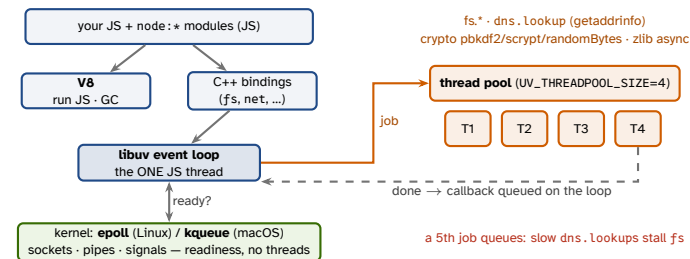
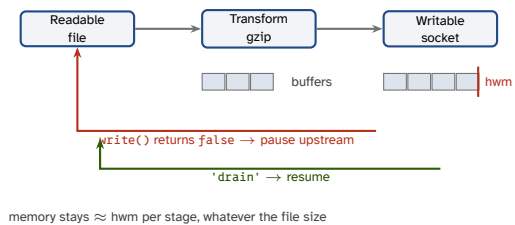


In one line: Node = **V8** (runs JS) + **libuv** (event loop, OS async I/O, a **4-thread pool**) + C++ **bindings**. Network I/O needs *no* threads (epoll / kqueue readiness); fs, dns, Lookup, crypto and zlib go to the pool. Your JS stays on **one thread** — block it and every request waits.

1 where the work happens



2 streams: backpressure



How it works

- OS async** for sockets: the kernel reports ready fds (poll phase); 10k idle connections cost memory, not threads.
- Thread pool** for what the OS cannot do asynchronously (regular files, getaddrinfo) or is CPU-heavy (crypto, zlib). Default **4**, max 1024; set `UV_THREADPOOL_SIZE` before first use. `dns.resolve*` uses c-ares on the network, not the pool.
- Blocking the loop:** *Sync APIs, `JSON.parse/stringify` of MBs, ReDoS regex, big sorts. Symptoms: p99 jumps on *all* routes, timers late, health checks time out, *one* core at 100%. Diagnose: `perf_hooks.monitorEventLoopDelay()`, `--cpu-prof` flame graph. Fix: stream, chunk + yield, or a worker.
- Streams:** Readable, Writable, Duplex (socket), Transform (zlib). Buffer per stream = `highWaterMark`: 64 KiB for bytes (16 KiB before Node 22), 16 objects in objectMode. `pipeline()` wires backpressure *and* error + destroy of every stage; `.pipe()` forwards no errors (leaks fds). Readables are for *await*-able.
- Buffer:** a `Uint8Array` subclass, memory outside the V8 heap (`memoryUsage().external`). `alloc` is zeroed; `allocUnsafe` may hold old bytes. Encodings: utf8, hex, base64(url), latin1, utf16le. A UTF-8 char can split across chunks: `setEncoding('utf8')`, not `chunk.toString()`.
- EventEmitter:** emit is **synchronous**, in registration order; 'error' with no listener **throws**; >10 listeners for one event → `MaxListenersExceededWarning` (usually a leak).

Example — stream, iterate, shut down

```
import { pipeline } from 'node:stream/promises';
await pipeline(createReadStream('big.log'), createGzip(),
  createWriteStream('big.log.gz')); // bounded memory + cleanup
for await (const chunk of req) size += chunk.length;
process.on('SIGTERM', () => { // k8s / systemd stop
  server.close(async () => { // no new conns, finish old
    await db.end(); process.exit(0); });
  setTimeout(() => process.exit(1), 10_000).unref(); // deadline
});
```

Scaling past one thread

	isolation / sharing	talk	for
worker_threads	own V8 isolate + loop, same process; <code>SharedArrayBuffer</code> + <code>Atomics</code>	<code>postMessage</code> (clone), <code>transfer</code> <code>ArrayBuffer</code>	CPU work (pool them)
cluster	N processes, no shared memory; primary shares the listen port (round-robin)	IPC messages	use all cores for a server
child_process	any program; spawn streams, exec = shell + buffer, fork = Node + IPC	stdio / IPC	tools, isolation

Errors, crashes, modules

- unhandledRejection** crashes the process by default since **Node 15** (`--unhandled-rejections=throw`). **uncaughtException**: log and exit — state is unknown; let the supervisor restart.
- ESM**: "type": "module" makes .js ESM; .mjs is always ESM, .cjs always CJS. ESM: full file extensions, top-level `await`, no `__dirname` (`import.meta.url`). `require()` of sync ESM: unflagged in Node 22.12.

Node vs Deno vs Bun

	Node	Deno	Bun
engine	V8	V8	JavaScriptCore
core in	C++ + libuv	Rust + Tokio	Zig
TS	strips types (23.6+)	native	native
security	all access; opt-in <code>--permission</code>	deny by default , <code>--allow-net ...</code>	all access
npm	native	npm: specifiers	npm-compatible

Interview traps

- ! "Node is single-threaded" — *your JS* is; libuv, V8's GC and the pool are not.
- ! Raising the pool size does not fix CPU-bound JS on the loop — only a worker does.
- ! `emit()` is not async: a slow listener blocks the emitter's caller.
- ! Cluster/processes share nothing: sessions and rate limits need Redis/DB.

Remember

Kernel for sockets, pool for files, workers for CPU — and never block the one loop.

Likely questions

- Why is one slow endpoint slowing all? — sync CPU on the loop; profile, move to a worker.
- Backpressure? — `write()` false → stop reading until 'drain'; pipeline does it.
- iOS analogue? — loop ≈ main thread, pool ≈ a 4-wide GCD queue, workers ≈ isolated actors.