

Standalone n8n: Explained Simply (Full Detail)


Why a single n8n instance struggles under 200 VUs — Burger Shop Analogy

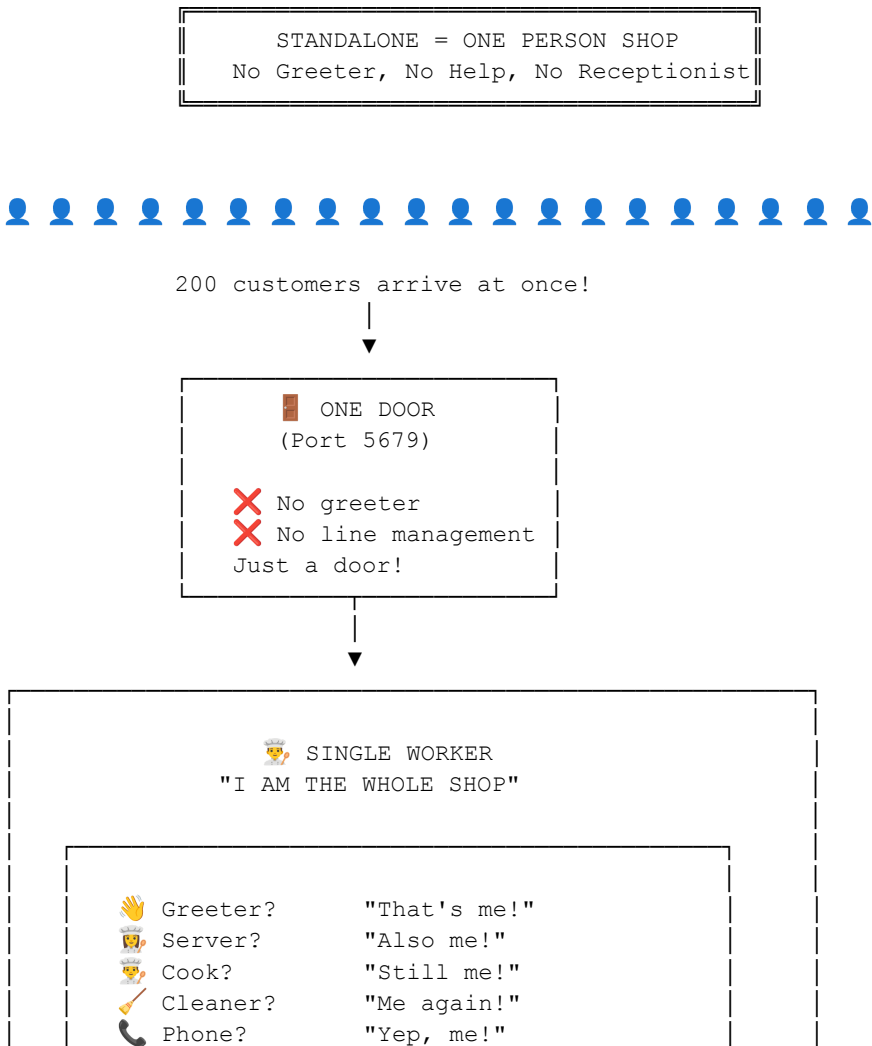
1) Big Idea (In One Sentence)

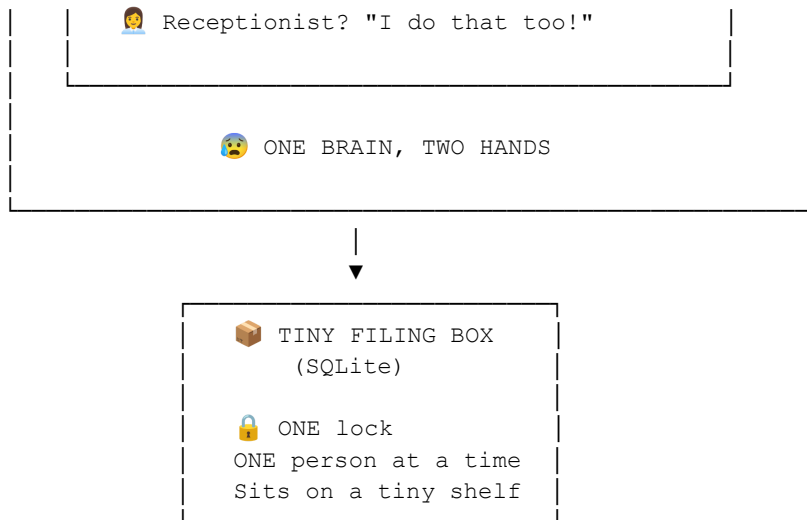
Standalone n8n is like a one-person burger shop: one process handles the door, takes orders, cooks, writes to the database, serves responses, and cleans up memory. Under heavy load, the single worker gets maxed out, the tiny filing box (SQLite) locks up, and most customers leave with errors.

2) Architecture Diagram

Standalone n8n: Explained Simply (Full Detail)

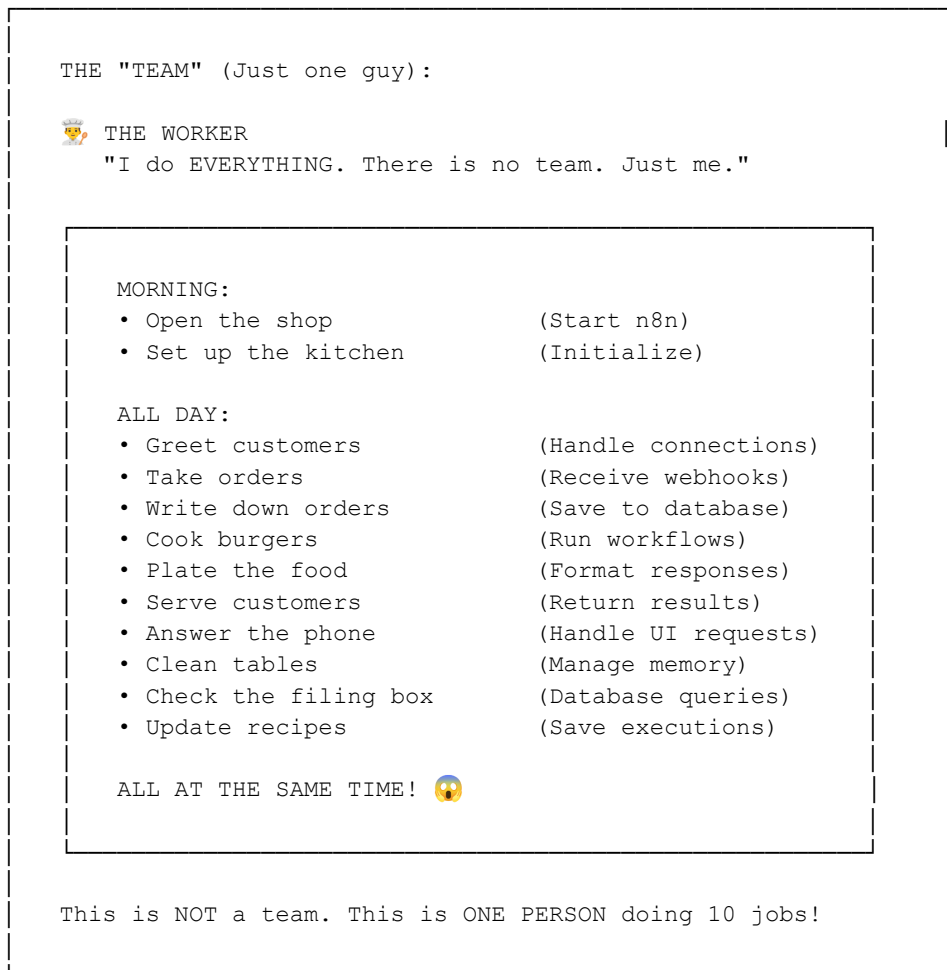
 Imagine a ONE-PERSON Burger Shop





3) The ONE Person Has to Do EVERYTHING

🎯 The ONE Person Has to Do EVERYTHING



4) Why CPU Hits 112% (The Single Brain + Single Thread)

Your machine may have many CPU cores, but a single Node.js process is fundamentally centered on one main event loop. Under heavy inbound traffic, that loop becomes the bottleneck: it must accept requests, schedule work, run JavaScript logic, coordinate database I/O, and serialize responses — all in one place.

🧠 Why CPU is 112%: The Single Brain Problem

💡 UNDERSTANDING CPU PERCENTAGE

Your computer has multiple "brains" (CPU cores)

🧠 Core 1	🧠 Core 2	🧠 Core 3	🧠 Core 4
[100%]	[100%]	[100%]	[100%]

Total possible = 400% (4 cores × 100%)

🚨 THE PROBLEM: Node.js is SINGLE-THREADED

This means n8n can only use ONE brain at a time!

🧠 Core 1	🧠 Core 2	🧠 Core 3	🧠 Core 4
[██████████ 112%]	[░░░░░░░░ 0%]	[░░░░░░░░ 0%]	[░░░░░░░░ 0%]



n8n is STUCK on one core!
Can't use the other 3 brains!

112% = One core MAXED OUT (100%) + a tiny bit of overflow

🍔 BURGER SHOP TRANSLATION:

Imagine you have TWO HANDS but can only use ONE:

👉 LEFT HAND: [██████████] 112% BUSY
"I'm flipping burgers, taking orders,
answering phone, ALL WITH ONE HAND!"

👉 RIGHT HAND: [░░░░░░░░] 0% (tied!)
"I could help but I'm not allowed!"

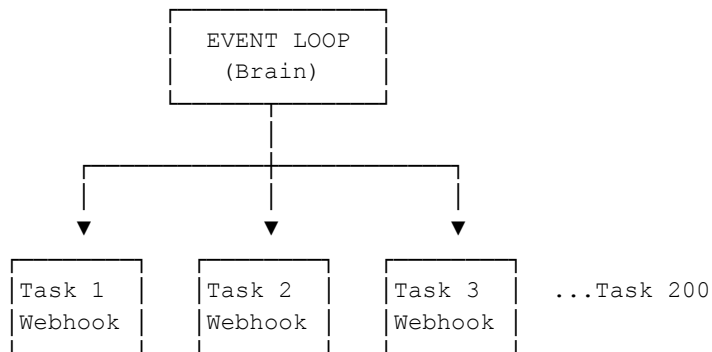
Single-threaded = Working with one hand tied behind back!

5) The Event Loop (Why Latency Spikes Under Load)

The event loop is like a spinning to-do list. When too many tasks arrive, everything queues up. Even if some steps are I/O-based, the coordination work, callbacks, serialization, and framework overhead still run through the same loop.

🧠 The Event Loop: Why Things Get Stuck

Node.js uses an "EVENT LOOP" - like a spinning to-do list



The brain can only do ONE TASK at a time!

🧠 "OK, Task 1... done! Task 2... done! Task 3..."

Meanwhile:

Task 50: "HELLO? I'M WAITING!"

Task 100: "STILL HERE!"

Task 200: "I'VE BEEN WAITING FOREVER!" 😡

🍔 BURGER SHOP TRANSLATION:

👤 Customer 1: "I want a burger!"

👨🍳 "OK, let me take your order, cook it, serve it..."

👤👤👤👤👤👤 Customers 2-200: *waiting in line*

👤 Customer 200: "I've been here for 45 SECONDS!"

One person = One customer at a time = LONG WAITS!

6) The Tiny Filing Box Problem (SQLite Locks)

SQLite is perfect for light/local usage, but under high write/read contention it behaves like a single locked file. When the process needs to write execution data or read state, locking can cascade into delays and failures.

📦 The Tiny Filing Box Problem (SQLite)

SQLITE = A tiny box that only ONE person can use at a time

📦 TINY BOX

🔒 LOCKED



recipes



👷 Worker

"Let me check
the recipe..."

THE PROBLEM:

Step 1: 👷 "I need to READ a recipe"
📦 🔒 *LOCKED* while reading

Step 2: 👷 "Now I need to WRITE an order"
📦 🔒 *LOCKED* while writing


Step 3: 👷 "Now I need to READ again"
📦 🔒 *LOCKED* again

Every time I touch the box, EVERYTHING WAITS!


SQLite locks the ENTIRE FILE for each operation!
At 200 customers, the box is locked almost constantly!

7) CPU Breakdown (Where the 112% Goes)

Under load, a surprisingly large portion of CPU goes to coordination overhead: handling incoming connections, scheduling tasks, serializing payloads, and repeated database interactions. Only part of it is 'real workflow work'.


 CPU Breakdown: Where Does 112% Go?

WHAT IS THE CPU DOING? (112% total)

 Handling connections (webhooks coming in)


[██] 30%

"200 people knocking on the door at once!"

 Event loop overhead (managing the queue)


[██] 20%

"Who's next? Who's next? Who's next?"

 SQLite operations (reading/writing)


[██] 30%

"Lock box, read, unlock, lock, write, unlock..."

 Actual workflow execution

[██] 20%

"Finally cooking burgers!"

 Garbage collection (cleaning up memory)

[██] 12%

"So much mess to clean, no time!"

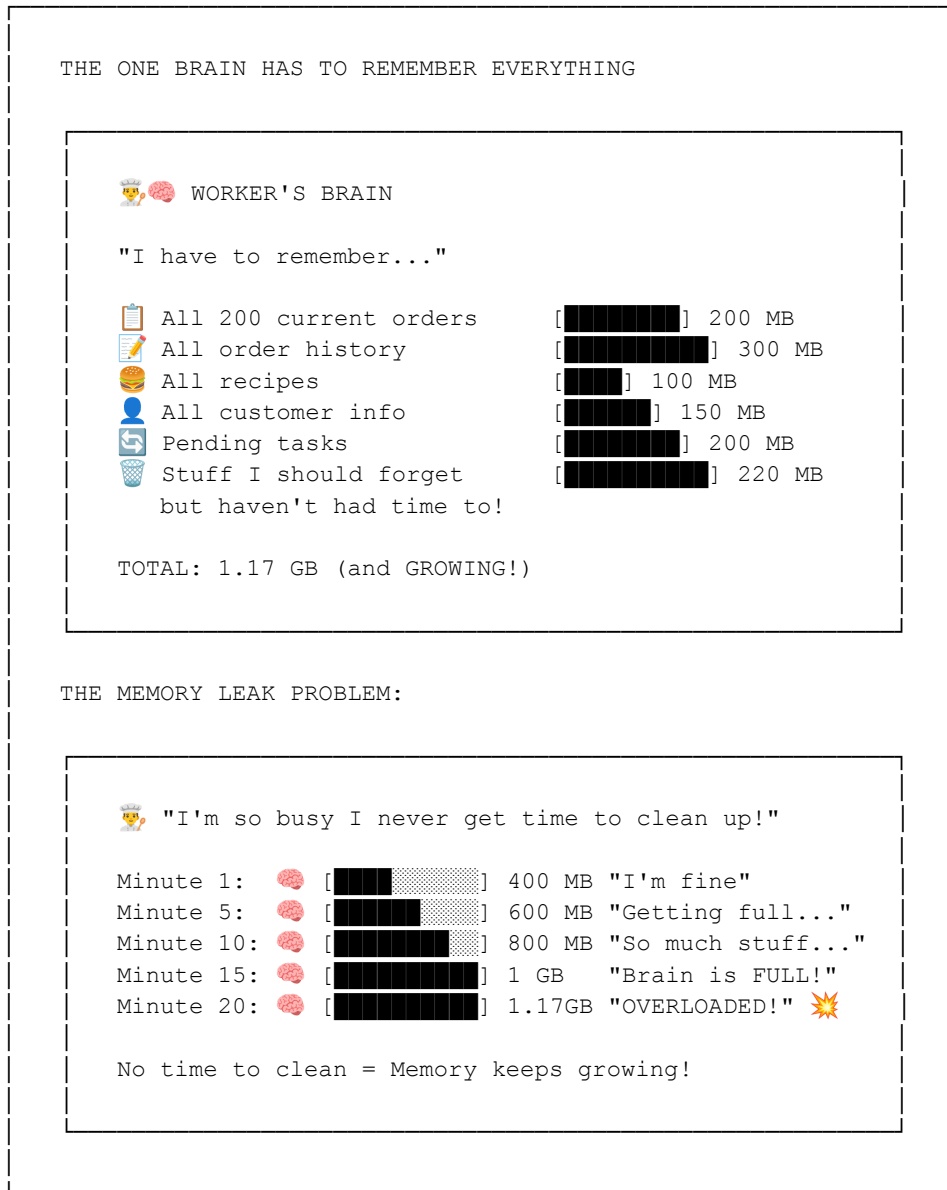
TOTAL: 112% (one core maxed + overflow)

 Only 20% is ACTUAL WORK! The rest is overhead!

8) Memory (Why It Climbs to 1.17 GB)

When one process holds everything (requests in flight, execution details, workflow state, UI traffic, logs, and database buffers), memory naturally climbs. If cleanup can't keep pace, the instance slows down and becomes more crash-prone.

🧠 Memory: Why 1.17 GB? (The Overloaded Brain)




9) What Happens at 200 Customers (The Crash)


💣 What Happens at 200 Customers (The Crash)

MINUTE 0: Shop Opens

 →  → 



 "200 customers! I got this... I think..."

CPU: [████████████████████] 100%




 SQLite: "Locking... unlocking... locking..."

MINUTE 1: Getting Overwhelmed

 "WE'RE WAITING!"


  "I can't go any faster!"


CPU: [████████████████████] 112% (MAXED!)


 SQLite: "TOO MANY REQUESTS!"   



Some customers starting to get errors...



MINUTE 2: The Breaking Point



 "DATABASE IS NOT READY!"



 "DATABASE IS NOT READY!"



 "DATABASE IS NOT READY!"



  "I can't even reach the filing box anymore!"



  "Error 503!"

  "Error 503!"

  "Error 503!"

  "Error 503!"

  "Error 503!"

  "Error 503!"

65% of customers getting REJECTED!

FINAL RESULT:

😊😊😊😊😊😊😊😊 35% served (12,996 requests)

😞😞😞😞😞😞😞😞😞😞😞😞😞😞😞😞 65% REJECTED
(23,657 errors!)

Longest wait: 45.53 SECONDS! 🤯

Shop status: 🦴 OVERWHELMED

10) Why Standalone Fails (Summary)

vs Why Standalone Fails (Summary)

❌ PROBLEM 1: SINGLE THREAD (One Hand Tied)

Computer: 🧠🧠🧠🧠 "I have 4 cores!"
Node.js: 🧠 "I can only use 1"
Result: 112% CPU = MAXED on one core, others unused

❌ PROBLEM 2: SQLITE (Tiny Locked Box)

📦 "Only ONE operation at a time!"
📦 "Everyone else must WAIT!"
Result: "Database not ready!" errors

❌ PROBLEM 3: NO SEPARATION (One Person Does All)

👨🍳 Takes orders + Cooks + Serves + Cleans + Everything
Result: Cannot handle multiple customers efficiently

❌ PROBLEM 4: NO MEMORY LIMITS (Brain Overflow)

🧠 No breaks, no cleanup, just keeps filling up
Result: 1.17 GB memory and growing

❌ PROBLEM 5: NO LOAD BALANCING (One Door)

🚪 Everyone comes through one entrance
Result: Bottleneck at the door

11) The Numbers Tell the Story

📊 The Numbers Tell the Story

STANDALONE AT 200 VUs

CPU: 112%
Memory: 1.17 GB
Success: 35%
Duration: ~2 minutes

vs PRODUCTION: CPU ~6% | Memory ~251 MB | Success 100% | 1h+

12) Bottom Line (One-Liner)

Standalone n8n works for light traffic, but at 200 VUs it collapses because one single-threaded process becomes the choke point while SQLite locking and memory growth amplify delays until failures dominate.