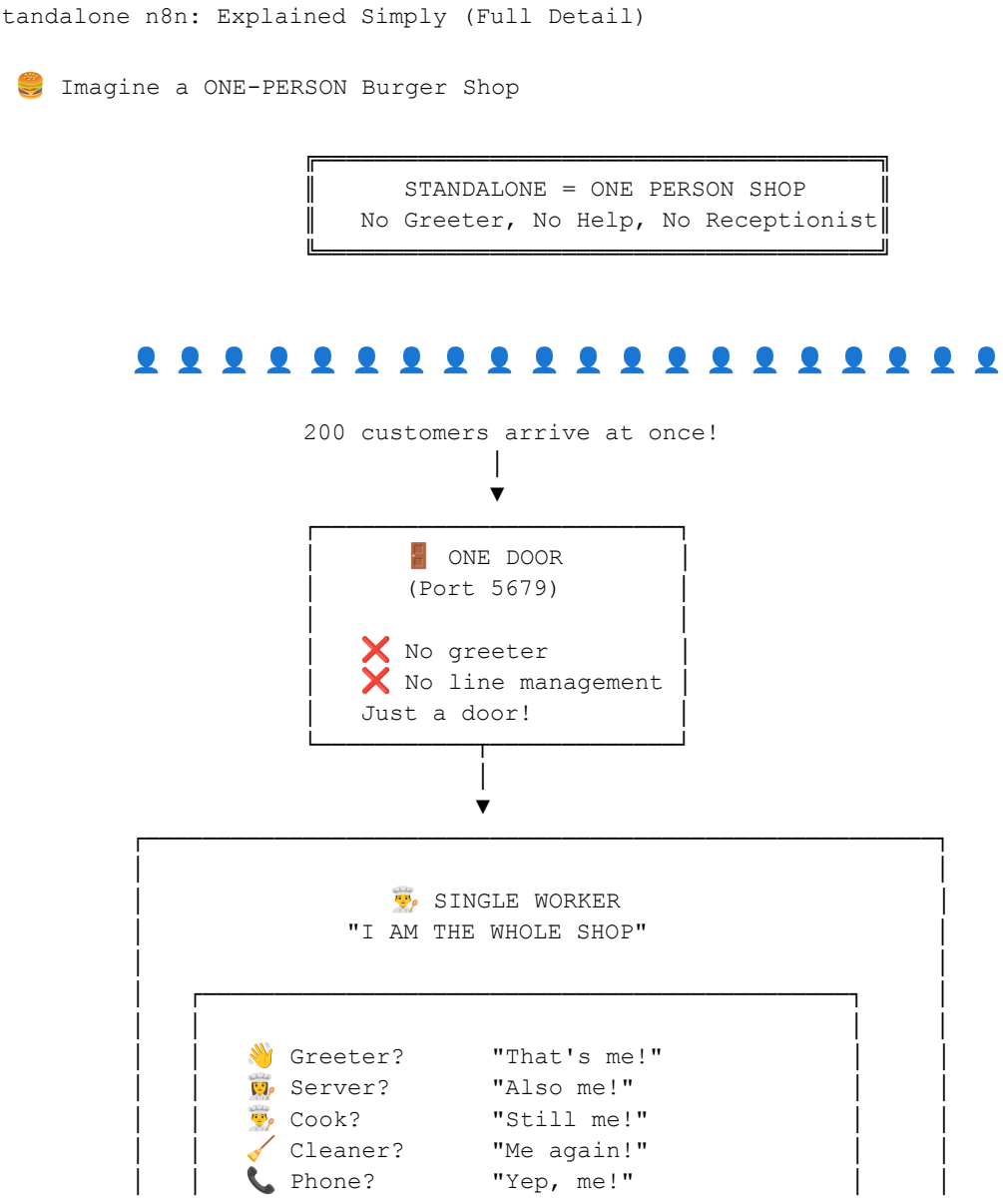
Standalone n8n: Explained Simply (Full Detail)

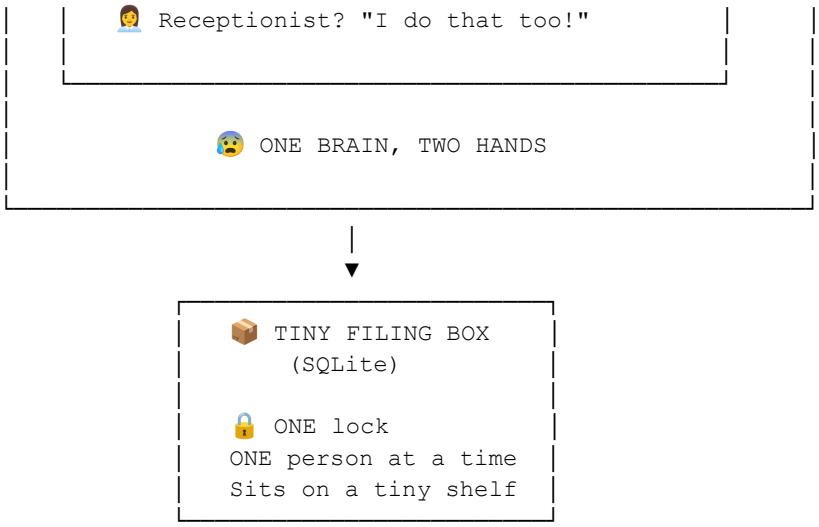
Why a single n8n instance struggles under 200 VUs — Burger Shop Analogy

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





3) The ONE Person Has to Do EVERYTHING

🎯 The ONE Person Has to Do EVERYTHING

THE "TEAM" (Just one guy):

👷 THE WORKER
"I do EVERYTHING. There is no team. Just me."

MORNING:

• Open the shop	(Start n8n)
• Set up the kitchen	(Initialize)

ALL DAY:

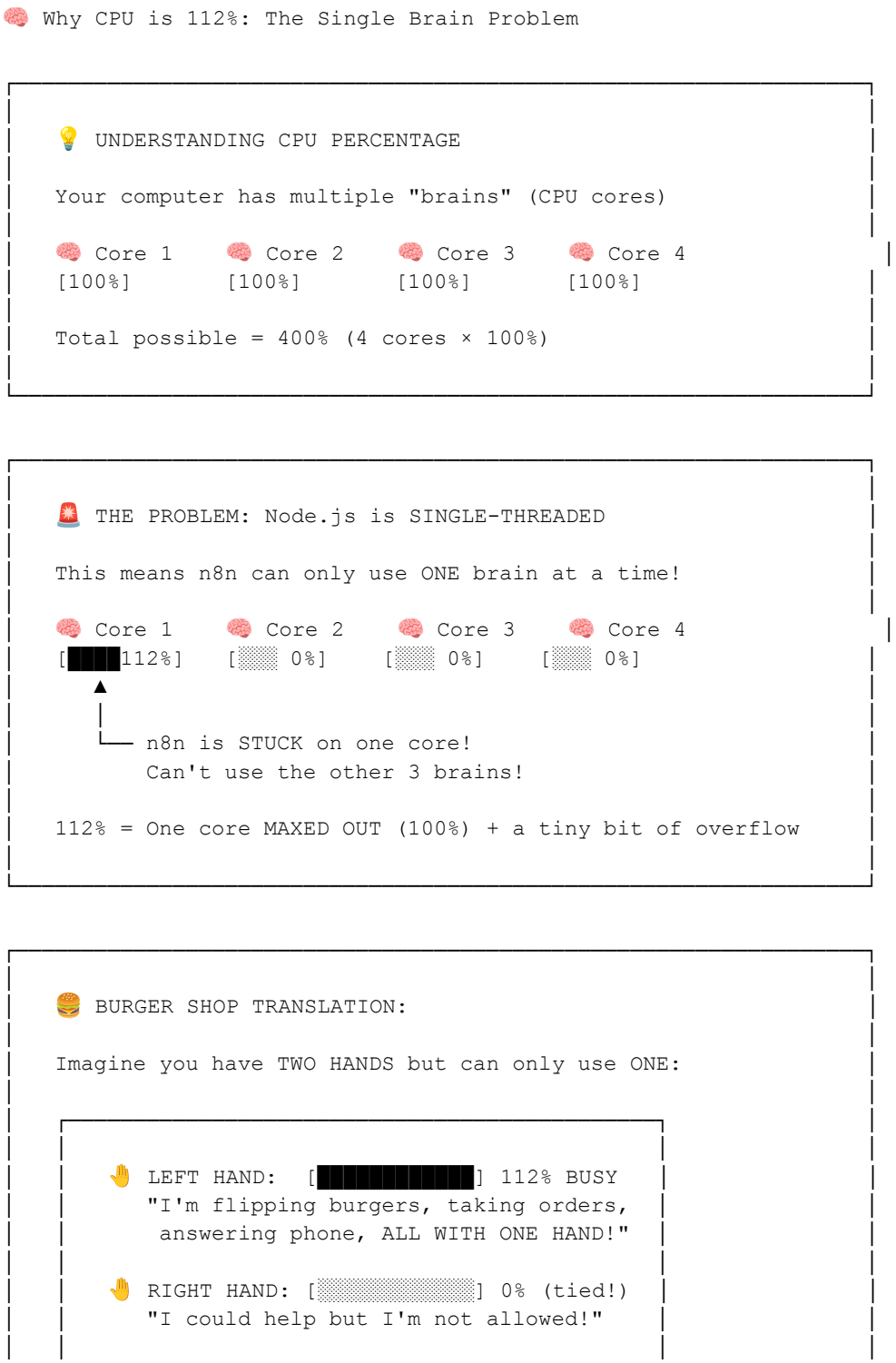
• Greet customers	(Handle connections)
• Take orders	(Receive webhooks)
• Write down orders	(Save to database)
• Cook burgers	(Run workflows)
• Plate the food	(Format responses)
• Serve customers	(Return results)
• Answer the phone	(Handle UI requests)
• Clean tables	(Manage memory)
• Check the filing box	(Database queries)
• Update recipes	(Save executions)

ALL AT THE SAME TIME! 😱

This is NOT a team. This is ONE PERSON doing 10 jobs!

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

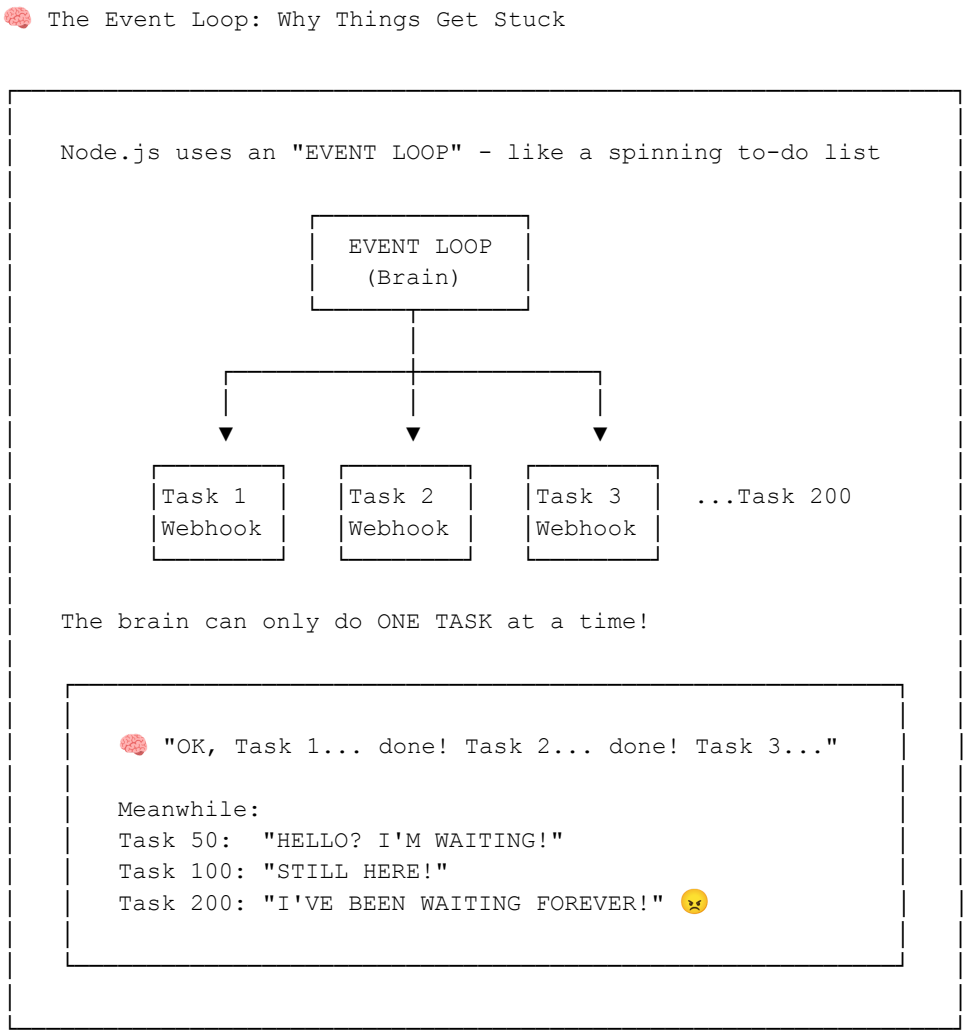
Your machine may have many CPU cores, but a single Node.js process is fundamentally centered on one main event loop. Under heavy inbound traffic, that loop becomes the bottleneck: it must accept requests, schedule work, run JavaScript logic, coordinate database I/O, and serialize responses — all in one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



```
└──────────────────────────────────────────────┘

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

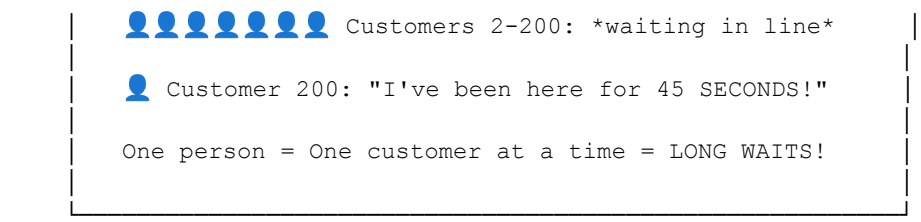
5) The Event Loop (Why Latency Spikes Under Load)

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.



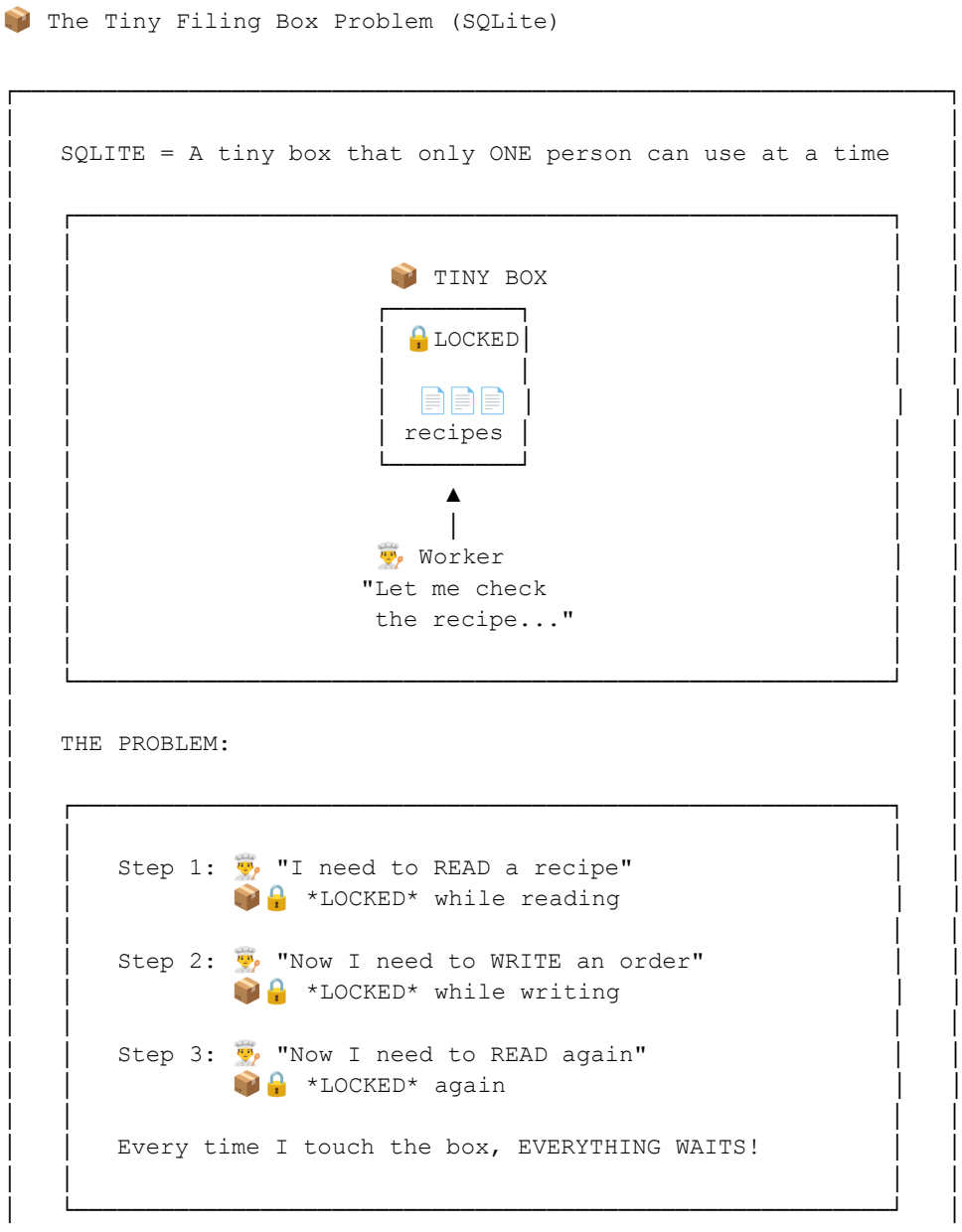
🍔 BURGER SHOP TRANSLATION:

```
┌──────────────────────────────────────────────┐
│                                              │
│  👤 Customer 1: "I want a burger!"           │
│  👨‍🍳 "OK, let me take your order, cook it, serve it..."│
│                                              │
```



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.



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

7) CPU Breakdown (Where the 112% Goes)

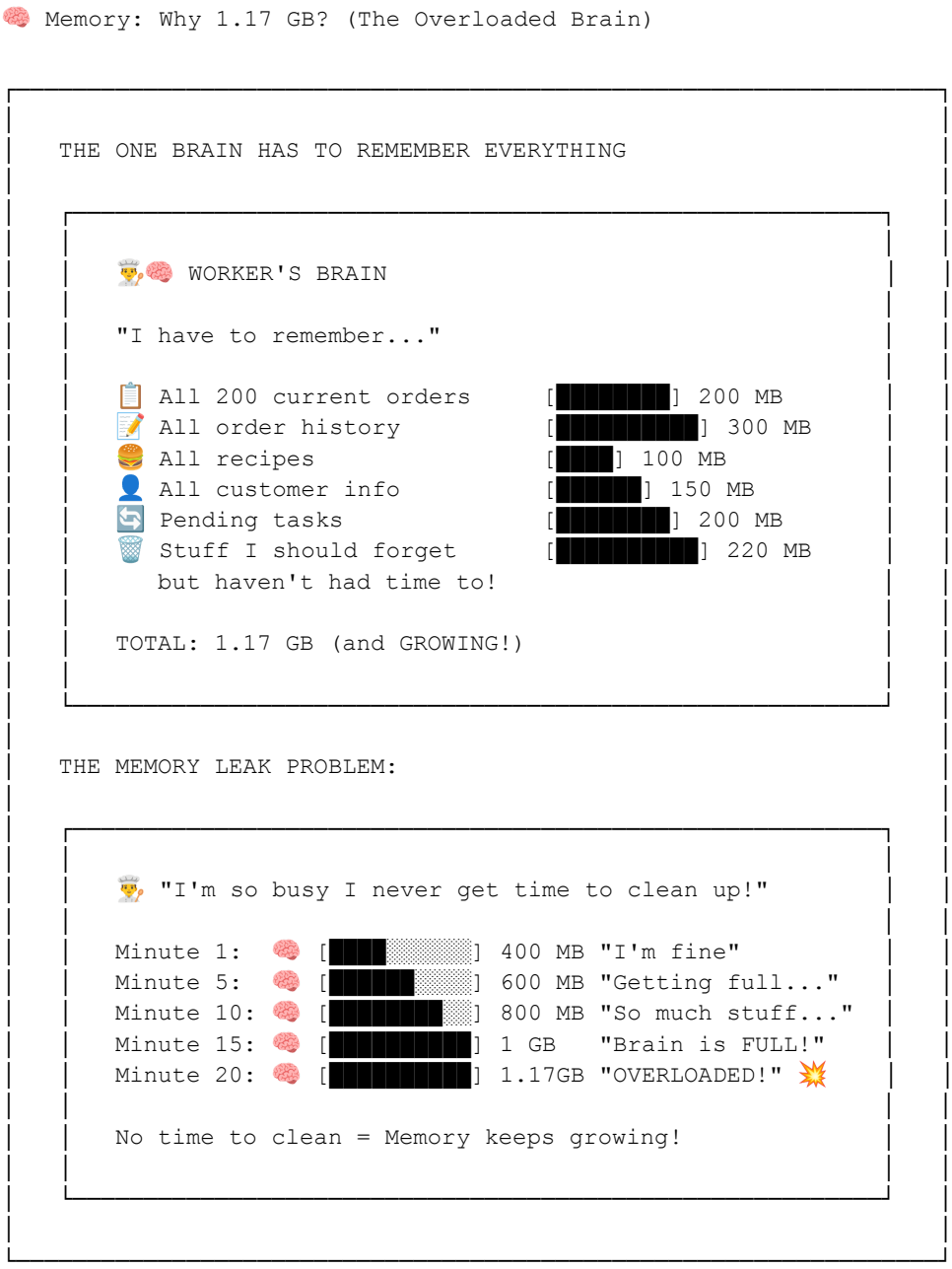
Under load, a surprisingly large portion of CPU goes to coordination overhead: handling incoming connections, scheduling tasks, serializing payloads, and repeated database interactions. Only part of it is 'real workflow work'.

```
CPU Breakdown: Where Does 112% Go?
```

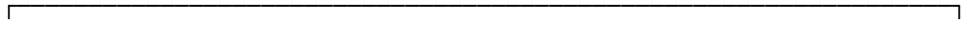
```
WHAT IS THE CPU DOING? (112% total)

  🌐 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!
```

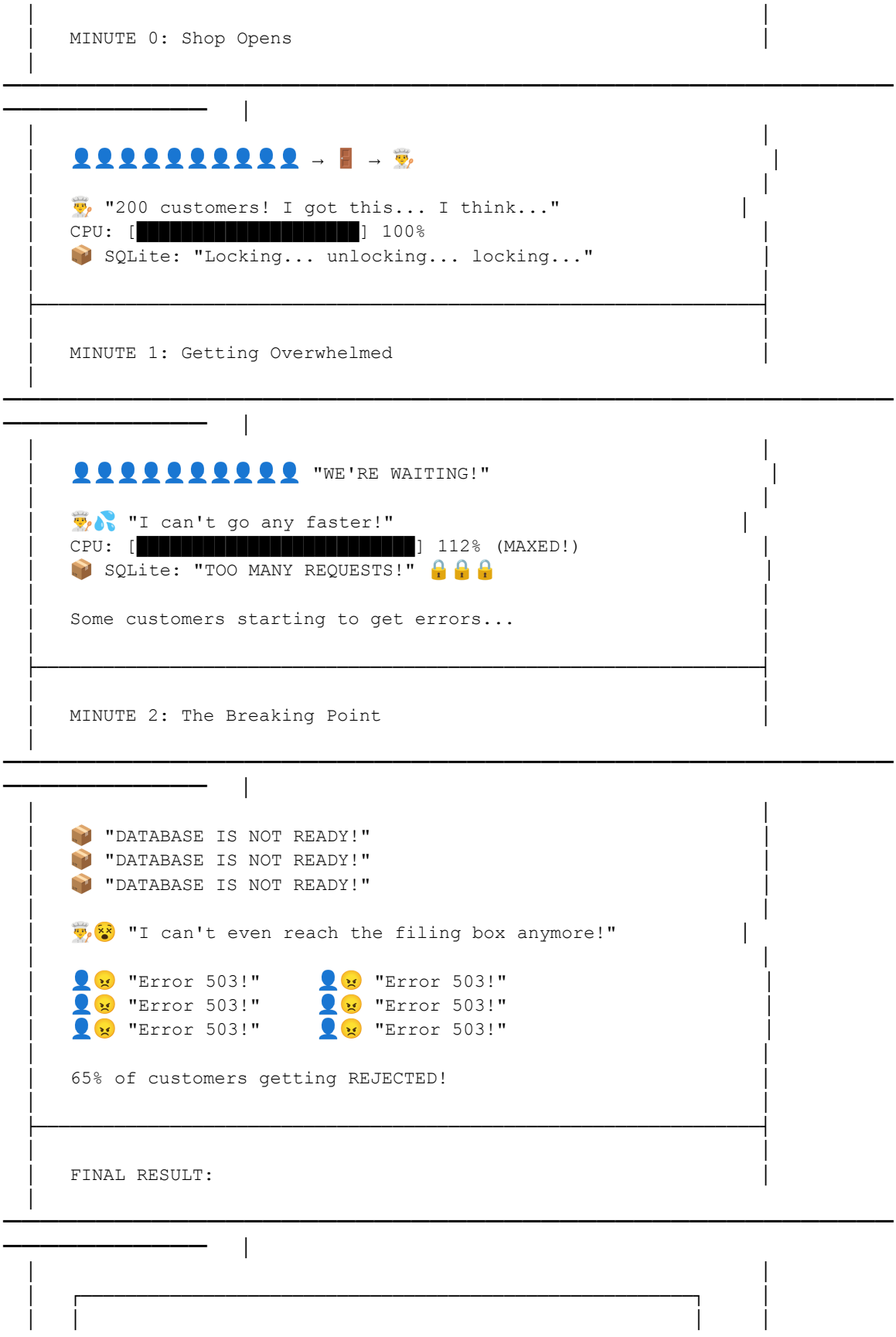
8) Memory (Why It Climbs to 1.17 GB)

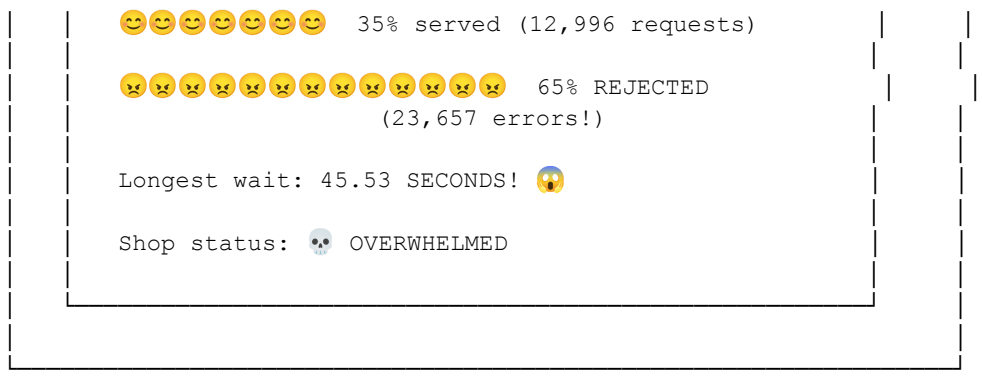
When one process holds everything (requests in flight, execution details, workflow state, UI traffic, logs, and database buffers), memory naturally climbs. If cleanup can't keep pace, the instance slows down and becomes more crash-prone.



9) What Happens at 200 Customers (The Crash)

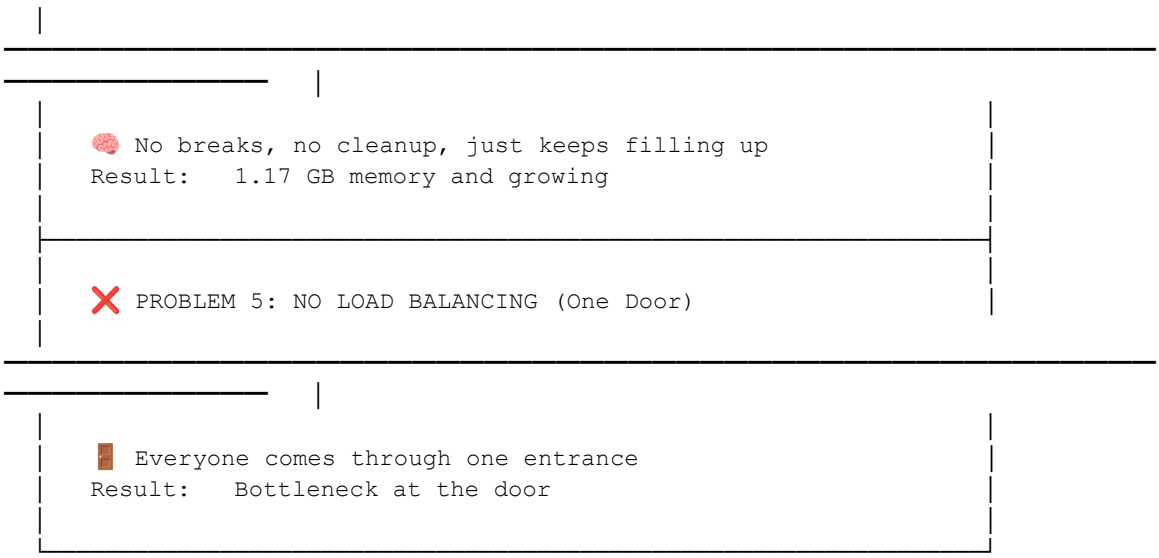



MINUTE 0: Shop Opens
👤👤👤👤👤👤👤👤 → 🚪 → 👩‍🍳
👩‍🍳 "200 customers! I got this... I think..."
CPU: [████████████████████] 100%
📦 SQLite: "Locking... unlocking... locking..."
MINUTE 1: Getting Overwhelmed
👤👤👤👤👤👤👤👤 "WE'RE WAITING!"
👩‍🍳💦 "I can't go any faster!"
CPU: [████████████████████████] 112% (MAXED!)
📦 SQLite: "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:



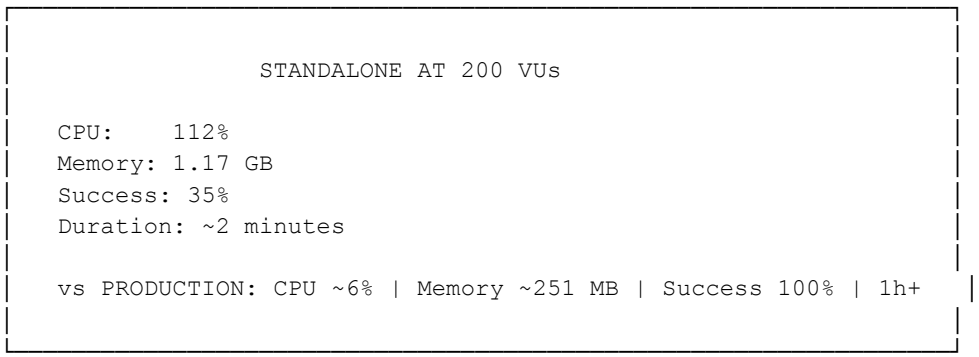
10) Why Standalone Fails (Summary)

🆚 Why Standalone Fails (Summary)



11) The Numbers Tell the Story

📊 The Numbers Tell the Story



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.