**🧠 What is Connection Pooling?**

**Connection pooling** is the practice of creating and reusing a fixed number of *ready-to-use* connections, rather than opening and closing a new one every time a request is made.

Imagine you're at a coffee shop:

* Without pooling: you stand in line, order, pay, get your drink, leave — and do this every single time you want a sip ☕.
* With pooling: you **reserve a table**, and every time you need more coffee, someone brings it **without restarting the whole process**.

**🗄 Part 1: Database Connection Pooling (AdonisJS + PostgreSQL)**

**🚫 Without pooling:**

Every API call (e.g. saving a prompt) has to:

1. Open a new socket to the database
2. Authenticate
3. Set up a session
4. Run the query
5. Tear down the connection

This is **slow** and can **overload your database** if 100 users hit your server at once — that's 100 new connections.

**✅ With pooling:**

* AdonisJS opens ~2–10 **shared connections** to PostgreSQL at boot
* These are **kept alive** and **ready to use**
* Every request simply “borrows” one for the query and returns it

🔁 No re-auth, no handshake, no TCP teardown — just reuse.

**💡 Analogy:**

It’s like having 10 trained workers ready to serve, instead of hiring and firing someone for every customer.

**🌐 Part 2: HTTP Connection Pooling (Axios ➜ LLaMA)**

**🚫 Without pooling:**

Each axios.post() to http://localhost:11434 opens a new HTTP connection:

1. TCP handshake (SYN → SYN-ACK → ACK)
2. Request + Response
3. Close socket

Each round takes ~20–50ms just in connection overhead 😩.

**✅ With keepAlive: true:**

* Axios reuses a **single warm TCP socket**
* The OS avoids handshaking over and over
* Latency drops to <1ms for sending the prompt

🔌 Less CPU, less memory, less garbage collection.

**📊 Real-world Benefits**

| **Feature** | **Without Pooling** | **With Pooling** |
| --- | --- | --- |
| DB performance | Slow under load (1:1 requests:connections) | Fast & efficient |
| HTTP requests | Frequent TCP churn | Persistent, low-latency |
| Scalability | Crashes under traffic | Handles spikes smoothly |
| Resource usage | Wasteful: many new sockets | Reuse: fewer, shared sockets |

**✅ Summary**

**Connection pooling** is about reusing expensive setup work:

* For **databases**, it avoids opening new DB connections each time.
* For **HTTP**, it avoids slow TCP handshakes and leverages socket reuse.

It’s one of the **lowest-effort, highest-impact** performance upgrades for APIs. You're now running a **professional-grade backend**.

Let me know if you want visual diagrams or want to see metrics with tools like pg\_stat\_activity or curl -v!

Set up proper connection pooling for API requests to improve performance.  
Below is all you need to add **connection-pooling** on both sides that matter:

| **Layer** | **What to change** | **Why it helps** |
| --- | --- | --- |
| **PostgreSQL → Lucid (Adonis)** | Tune the pool in config/database.ts | Re-uses a small set of DB sockets instead of opening / closing one per request |
| **Axios → LLaMA HTTP** | Re-use a keep-alive HTTP agent | Avoids a new TCP handshake for every prompt you forward to the local model |

1 ⋅ Database pool (Lucid + pg)

config/database.ts

1. import Env from '@adonisjs/core/env'

2.

3. export default {

4. /\*

5. |--------------------------------------------------------------------------

6. | Postgres connection

7. |--------------------------------------------------------------------------

8. \*/

9. pg: {

10. client: 'pg',

11. connection: {

12. host: Env.get('PG\_HOST'),

13. port: Env.get('PG\_PORT'),

14. user: Env.get('PG\_USER'),

15. password: Env.get('PG\_PASSWORD', ''),

16. database: Env.get('PG\_DB\_NAME'),

17. },

18.

19. /\*\* 👇 Pool settings \*/

20. pool: {

21. min: 2, // never drop below 2 idle connections

22. max: 10, // no more than 10 simultaneous

23. idleTimeoutMillis: 30\_000, // (default 10 s) – disconnect if idle 30 s

24. acquireTimeoutMillis: 10\_000, // fail fast if pool is exhausted

25. },

26. },

27. }

28.

**Tip:** If you expect spikes, set max ≈ *2 × CPU cores*; don’t leave it unbounded.

Nothing else to do – Lucid now keeps a warm pool and hands a client back instantly for each request.

**2 ⋅ Keep-alive HTTP agent for Axios ➜ LLaMA**

app/Services/LlmService.ts

1. import axios from 'axios'

2. import http from 'node:http'

3.

4. /\*\* 🔄 one keep-alive agent reused by every call \*/

5. const keepAliveAgent = new http.Agent({ keepAlive: true })

6.

7. export default class LlmService {

8. static async generateResponse(prompt: string): Promise<string> {

9. const { data } = await axios.post(

10. 'http://127.0.0.1:11434/api/generate',

11. { model: 'llama3', prompt, stream: false },

12. { httpAgent: keepAliveAgent } // 👈 reuse the same socket

13. )

14.

15. return data.response // adjust if your field differs

16. }

17. }

18.

* With keepAlive: true, Node holds the TCP socket open;
* Subsequent prompts ride the same connection – zero handshake latency (<1 ms vs ~20 ms).

**Optional Production Hardening**

| **Need** | **What to add** |
| --- | --- |
| **Very high throughput** | Run **pgBouncer** in front of Postgres and point PG\_PORT to it. |
| **Circuit breaker for LLM** | Use a package like cockatiel or @fastify/reply-from with a fallback. |
| **HTTPS LLM endpoint** | Replace the http.Agent with https.Agent({ keepAlive: true, rejectUnauthorized: false }). |