Ollama API Integration

Perfect — you already have the “non-streaming” setup working. Now you want:

1. **Show assistant output line-by-line as it arrives** (streaming UI)
2. **Persist last user query + last assistant reply in `localStorage`**, and restore on reload

To do (1) properly, you should **stream from Ollama** (`stream: true`) and **forward that stream** from Express to the browser using **SSE** (Server-Sent Events). The browser can then render incremental chunks immediately.

Below is the cleanest approach with minimal changes.

✅ Backend: add a streaming endpoint (SSE)

Create a new route in `server/routes/ai.js`:

`POST /api/ai/chat-stream`

- Calls Ollama with `stream: true`
- Reads the streamed JSON lines
- Sends them to the browser as SSE: `data: ...\n\n`

Replace your `server/routes/ai.js` with this (keep your existing `/chat` too):

```
js

// server/routes/ai.js
import express from "express";

const router = express.Router();

// existing non-stream endpoint (keep)
router.post("/chat", async (req, res) => {
  try {
    const { messages, model = "dolphin-llama3" } = req.body;

    if (!Array.isArray(messages)) {
      return res.status(400).json({ error: "messages must be an array" });
    }

    const ollamaRes = await fetch("http://127.0.0.1:11434/api/chat", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ model, messages, stream: false }),
    });

    if (!ollamaRes.ok) {
      const errText = await ollamaRes.text();
      return res.status(ollamaRes.status).json({ error: errText });
    }

    const data = await ollamaRes.json();
    return res.json({ reply: data?.message?.content ?? "", raw: data });
  } catch (e) {
    return res.status(500).json({ error: e?.message || "Server error" });
  }
});
```

```
// ✅ NEW: streaming endpoint (SSE)
router.post("/chat-stream", async (req, res) => {
  try {
    const { messages, model = "dolphin-llama3" } = req.body;

    if (!Array.isArray(messages)) {
      return res.status(400).json({ error: "messages must be an array" });
    }

    // SSE headers
    res.setHeader("Content-Type", "text/event-stream; charset=utf-8");
    res.setHeader("Cache-Control", "no-cache, no-transform");
    res.setHeader("Connection", "keep-alive");

    // If behind proxies, helps flush quickly
    res.flushHeaders?.();

    const ollamaRes = await fetch("http://127.0.0.1:11434/api/chat", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ model, messages, stream: true }),
    });

    if (!ollamaRes.ok || !ollamaRes.body) {
      const errText = await ollamaRes.text().catch(() => "Ollama error");
      res.write(`event: error\ndata: ${JSON.stringify({ error: errText })}\n\n`);
      return res.end();
    }

    const reader = ollamaRes.body.getReader();
    const decoder = new TextDecoder("utf-8");

    let buffer = "";

    while (true) {
      const { value, done } = await reader.read();
      if (done) break;

      buffer += decoder.decode(value, { stream: true });

      // Ollama sends JSON objects separated by newlines
      const lines = buffer.split("\n");
      buffer = lines.pop() || ""; // keep incomplete tail

      for (const line of lines) {
        if (!line.trim()) continue;

        // forward raw json line as SSE data
        res.write(`data: ${line}\n\n`);
      }
    }

    // send any remaining buffered line
    if (buffer.trim()) {
      res.write(`data: ${buffer}\n\n`);
    }

    res.write(`event: end\ndata: {}\n\n`);
    res.end();
  } catch (e) {
    res.write(`event: error\ndata: ${JSON.stringify({ error: e?.message || "Server error"
})}\n\n`);
    res.end();
  }
});

export default router;
```

✅ Backend CORS: allow streaming properly

Your CORS config is okay. But for streaming in browsers, it's good to allow OPTIONS too.

In `server/index.js`, **uncomment**:

```
js

app.options("*", cors());
```

So:

```
js

app.use(cors({ /* your config */ }));
app.options("*", cors());
```

✅ Frontend: render assistant reply progressively + localStorage

A) Add helper to update "last assistant bubble" as stream arrives

Add these helpers inside `<script>` (near your other functions):

```
js

const LS_KEY = "ollama_last_chat";

function saveLastToLocalStorage(userText, assistantText) {
  localStorage.setItem(LS_KEY, JSON.stringify({
    userText,
    assistantText,
    ts: Date.now()
  }));
}

function loadLastFromLocalStorage() {
  try {
    return JSON.parse(localStorage.getItem(LS_KEY) || "null");
  } catch {
    return null;
  }
}

// Create an assistant message bubble we can update live
function addAssistantStreamingBubble() {
  const msg = document.createElement("div");
  msg.className = "msg ai";

  const bubble = document.createElement("div");
  bubble.className = "bubble";

  const small = document.createElement("div");
  small.className = "small";
  small.textContent = "Assistant";
  bubble.appendChild(small);

  const body = document.createElement("div");
  body.className = "text-block";
  body.textContent = ""; // will fill progressively
  bubble.appendChild(body);

  msg.appendChild(bubble);
  chatEl.appendChild(msg);
  chatEl.scrollTop = chatEl.scrollHeight;
```

```
  return { bodyEl: body, bubbleEl: bubble };
}
```

B) Replace your `send()` with a streaming version

This version:

- pushes user message to **`messages`**
- calls **`/api/ai/chat-stream`**
- reads SSE stream line by line
- appends tokens to the assistant output immediately
- keeps a final **`assistantText`** string
- saves last user/assistant to localStorage

Replace your **`send()`** with:

```
js

async function send() {
  const userText = promptEl.value.trim();
  if (!userText) return;

  sendBtn.disabled = true;
  statusEl.textContent = "thinking…";

  const prompt1 =
    userText +
    "\n\n" +
    "Explain like you're teaching a beginner developer, but keep it technically correct in 300
words. " +
    "Requirements: Start with a simple definition in 1–2 lines. Use an analogy. Give 3 code
examples. " +
    "Include var-for-loop setTimeout gotcha and fix using let. After each code block, explain in
bullets. " +
    "End with a one-line definition to memorize. Use headings.";

  // store chat history for context
  messages.push({ role: "user", content: prompt1 });

  // show user's original input
  addMessage("user", userText);
  promptEl.value = "";

  // create assistant bubble that updates progressively
  const { bodyEl, bubbleEl } = addAssistantStreamingBubble();
  let assistantText = "";

  try {
    const apiUrl = apiUrlEl.value.trim().replace(/\/chat$/, "/chat-stream");
    const model = modelEl.value.trim() || "dolphin-llama3";

    const res = await fetch(apiUrl, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ messages, model }),
    });

    if (!res.ok || !res.body) {
      const err = await res.text().catch(() => "");
      bodyEl.textContent = `Error: ${err || "Streaming failed"}`;
      statusEl.textContent = "error";
      return;
```

```
    }

    const reader = res.body.getReader();
    const decoder = new TextDecoder("utf-8");

    let buffer = "";

    while (true) {
      const { value, done } = await reader.read();
      if (done) break;

      buffer += decoder.decode(value, { stream: true });

      // SSE events separated by double newline
      const parts = buffer.split("\n\n");
      buffer = parts.pop() || "";

      for (const part of parts) {
        // We only care about lines that start with "data:"
        const lines = part.split("\n");
        for (const line of lines) {
          if (!line.startsWith("data:")) continue;
          const jsonStr = line.slice(5).trim();
          if (!jsonStr) continue;

          // Ollama stream sends JSON objects per line
          let obj;
          try { obj = JSON.parse(jsonStr); } catch { continue; }

          // In Ollama chat streaming, content is usually here:
          const chunk = obj?.message?.content ?? "";
          if (chunk) {
            assistantText += chunk;

            // ✅ "one by one line" effect:
            // Update the visible text as it grows
            bodyEl.textContent = assistantText;

            chatEl.scrollTop = chatEl.scrollHeight;
          }

          // done flag ends stream
          if (obj?.done) {
            break;
          }
        }
      }
    }

    // Save assistant msg into history for next turn context
    messages.push({ role: "assistant", content: assistantText });

    // Save last Q/A to localStorage
    saveLastToLocalStorage(userText, assistantText);

    // Now re-render the assistant message with code-block UI
    // (because streaming used plain text-block; we want code cards)
    // Replace streaming text with your fancy renderer:
    const rendered = renderWithCodeBlocks(assistantText);
    // remove old bodyEl and replace
    bodyEl.replaceWith(rendered);

    if (window.Prism) Prism.highlightAllUnder(bubbleEl);
    statusEl.textContent = "idle";
  } catch (e) {
    bodyEl.textContent = `Error: ${e?.message || "Network error"}`;
    statusEl.textContent = "error";
  } finally {
    sendBtn.disabled = false;
  }
}
```

Why we “re-render” at the end

During streaming, we show plain text quickly.
When done, we re-run your `renderWithCodeBlocks()` so code blocks become the **nice UI cards**.

C) Restore last chat on page load

Add this near the bottom (after your welcome message OR instead of it):

```
js

const last = loadLastFromLocalStorage();
if (last?.userText && last?.assistantText) {
  addMessage("user", last.userText);
  addMessage("assistant", last.assistantText);
} else {
  addMessage(
    "assistant",
    "Hi! Ask me anything. If I return code blocks, they'll appear in a code card with a Copy
button.\n\nExample:\n```js\nconsole.log('hello')\n```"
  );
}
```

Notes / gotchas

1) "One by one line" vs "token by token"

Ollama usually streams in small chunks (tokens/phrases), not always whole lines.
Your UI will update continuously. If you want to **force line-by-line**, you can buffer until `\n` and then append lines — but token streaming feels nicer.

2) The frontend must call `/chat-stream`

I made the code auto-replace `/chat` with `/chat-stream`, so you don't need to change the input field.

3) CORS preflight

If you see preflight issues again, keep:

```
js

app.options("*", cors());
```

If you want the **assistant response to appear as separate `<div>` lines** (each new line becomes its own block), tell me and I'll tweak the streaming renderer to append line elements instead of updating a single text node.