

# AI Agent Chatbot — Full-Stack Starter (NodeJS API + Python Agent + React Frontend)

This is a minimal, working scaffold for an AI-agent chatbot like ChatGPT using:

- **NodeJS (Express)** — API that simulates a DB and exposes data endpoints.
- **Python (FastAPI)** — AI Agent: routes user tasks to an LLM, calls Node API tools, formats results, and **streams** tokens.
- **React (Vite)** — Frontend chat UI with **typing-style streaming** like ChatGPT.
- **Docker Compose** — optional, to run everything together.

---

## 1) Project Structure

```
ai-agent-chatbot/  
├─ docker-compose.yml  
├─ node-api/  
│   ├─ package.json  
│   ├─ server.js  
│   └─ data.json  
├─ py-agent/  
│   ├─ requirements.txt  
│   ├─ main.py  
│   └─ .env.example  
└─ web-client/  
    ├─ index.html  
    ├─ package.json  
    ├─ vite.config.js  
    └─ src/  
        ├─ main.jsx  
        ├─ App.jsx  
        ├─ api.js  
        └─ styles.css
```

---

## 2) Docker Compose (optional but recommended)

```
# docker-compose.yml  
version: '3.9'  
services:  
  node-api:  
    build: ./node-api  
    ports: ["4000:4000"]  
    environment:
```

```

    - NODE_ENV=production
networks: [appnet]

py-agent:
  build: ./py-agent
  environment:
    - OPENAI_API_KEY=${OPENAI_API_KEY}
    - NODE_API_BASE_URL=http://node-api:4000
    - AGENT_HOST=0.0.0.0
    - AGENT_PORT=8000
  ports: ["8000:8000"]
  depends_on: [node-api]
  networks: [appnet]

web-client:
  build: ./web-client
  ports: ["5173:5173"]
  environment:
    - VITE_AGENT_BASE_URL=http://localhost:8000
  depends_on: [py-agent]
  networks: [appnet]

networks:
  appnet:
    driver: bridge

```

Put your `OPENAI_API_KEY` in your host shell env before `docker compose up --build`.

### 3) Node API (Express) — `node-api/`

`package.json`

```

{
  "name": "node-api",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.19.2"
  }
}

```

`data.json` — pretend DB data

```

{
  "products": [
    { "id": 1, "name": "Anycubic PLA Pro", "category": "filament", "price": 1299 },
    { "id": 2, "name": "Nozzle 0.4mm Brass", "category": "spare", "price": 199 },
    { "id": 3, "name": "Kobra 2 Neo Bed Sheet", "category": "accessory", "price": 899 },
    { "id": 4, "name": "PETG Transparent", "category": "filament", "price": 1499 }
  ],
  "users": [
    { "id": 101, "name": "Aakey", "tier": "pro" },
    { "id": 102, "name": "Guest", "tier": "free" }
  ]
}

```

server.js

```

import express from 'express';
import cors from 'cors';
import fs from 'fs';

const app = express();
app.use(cors());
app.use(express.json());

const raw = fs.readFileSync('./data.json', 'utf-8');
const db = JSON.parse(raw);

// Health
app.get('/health', (req, res) => res.json({ ok: true }));

// Products with optional filters: category, maxPrice
app.get('/products', (req, res) => {
  const { category, maxPrice } = req.query;
  let items = db.products;
  if (category) items = items.filter(p => p.category === category);
  if (maxPrice) items = items.filter(p => p.price <= Number(maxPrice));
  res.json({ items });
});

// User by id
app.get('/users/:id', (req, res) => {
  const id = Number(req.params.id);
  const user = db.users.find(u => u.id === id);
  if (!user) return res.status(404).json({ error: 'User not found' });
  res.json({ user });
});

```

```
const PORT = process.env.PORT || 4000;
app.listen(PORT, () => console.log(`Node API running on port ${PORT}`));
```

#### Dockerfile (optional)

```
FROM node:20-alpine
WORKDIR /app
COPY package.json package-lock.json* ./
RUN npm ci --omit=dev || npm i --omit=dev
COPY . .
EXPOSE 4000
CMD ["npm", "start"]
```

## 4) Python AI Agent (FastAPI) — `py-agent/`

`requirements.txt`

```
fastapi==0.111.0
uvicorn[standard]==0.30.0
httpx==0.27.0
python-dotenv==1.0.1
pydantic==2.7.4
openai==1.30.1
```

`.env.example`

```
OPENAI_API_KEY=sk-...
NODE_API_BASE_URL=http://localhost:4000
AGENT_HOST=0.0.0.0
AGENT_PORT=8000
```

`main.py`

```
import os
import json
import asyncio
from typing import AsyncGenerator, Dict, Any

import httpx
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse, StreamingResponse
from pydantic import BaseModel
```

```

from dotenv import load_dotenv

load_dotenv()

OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
NODE_API = os.getenv("NODE_API_BASE_URL", "http://localhost:4000")
HOST = os.getenv("AGENT_HOST", "0.0.0.0")
PORT = int(os.getenv("AGENT_PORT", "8000"))

app = FastAPI(title="AI Agent")

class ChatBody(BaseModel):
    message: str
    user_id: int | None = None

async def call_node_api(tool: str, params: Dict[str, Any]) -> Dict[str, Any]:
    async with httpx.AsyncClient(timeout=15.0) as client:
        if tool == "get_products":
            r = await client.get(f"{NODE_API}/products", params=params)
            r.raise_for_status()
            return r.json()
        if tool == "get_user":
            uid = params.get("id")
            r = await client.get(f"{NODE_API}/users/{uid}")
            r.raise_for_status()
            return r.json()
        return {"error": "unknown_tool"}

# ===== Minimal agent logic =====
# Uses OpenAI function-calling to decide which tool(s) to call, then formats
# a final answer.

from openai import OpenAI
client = OpenAI(api_key=OPENAI_API_KEY) if OPENAI_API_KEY else None

TOOLS = [
    {
        "type": "function",
        "function": {
            "name": "get_products",
            "description": "Fetch products with optional category and
maxPrice filters",
            "parameters": {
                "type": "object",
                "properties": {
                    "category": {"type": "string", "description": "Product
category"},
                    "maxPrice": {"type": "number", "description": "Max price
in INR"}
                },
            },
            "additionalProperties": False
        }
    },

```

```

        }
    }
},
{
    "type": "function",
    "function": {
        "name": "get_user",
        "description": "Fetch a user profile by ID",
        "parameters": {
            "type": "object",
            "properties": {
                "id": {"type": "integer"}
            },
            "required": ["id"],
            "additionalProperties": False
        }
    }
}
]

async def llm_route_and_answer(message: str, user_id: int | None) -> str:
    # If no OpenAI key, fallback to simple rule-based handling.
    if client is None:
        if "product" in message.lower():
            data = await call_node_api("get_products", {})
            items = data.get("items", [])
            lines = ["Products:"] + [f"- {p['name']} - ₹{p['price']}"
                                   for p in items]
            return "\n".join(lines)
        if "user" in message.lower() and user_id:
            data = await call_node_api("get_user", {"id": user_id})
            u = data.get("user")
            return f"User: {u['name']} (tier: {u['tier']})" if u else "User
not found"
        return "I can fetch products or a user by ID. Try: 'Show me filament
under 1400'"

    # With LLM: ask it to decide which tool to call.
    msgs = [
        {"role": "system", "content": "You are an assistant that can call
tools to fetch data, then summarize clearly in markdown."},
        {"role": "user", "content": message}
    ]

    resp = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=msgs,
        tools=TOOLS,
        tool_choice="auto"
    )

```

```

tool_calls = resp.choices[0].message.tool_calls or []
tool_results = []

for tc in tool_calls:
    fn = tc.function.name
    args = json.loads(tc.function.arguments or "{}")
    if fn == "get_products":
        # convert maxPrice to number safely
        if "maxPrice" in args and isinstance(args["maxPrice"], str) and
args["maxPrice"].isdigit():
            args["maxPrice"] = float(args["maxPrice"]) # pydantic will
coerce
        data = await call_node_api("get_products", args)
        tool_results.append({"tool": fn, "output": data})
    elif fn == "get_user":
        if not args.get("id") and user_id:
            args["id"] = user_id
        data = await call_node_api("get_user", args)
        tool_results.append({"tool": fn, "output": data})

# Ask LLM to summarize all tool results for the user.
msgs.extend([
    {"role": "tool", "name": tr["tool"], "content":
json.dumps(tr["output"]) } for tr in tool_results
])

final = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=msgs,
    temperature=0.2
)

return final.choices[0].message.content or "(no content)"

@app.post("/chat")
async def chat(body: ChatBody):
    text = await llm_route_and_answer(body.message, body.user_id)
    return JSONResponse({"answer": text})

@app.post("/chat/stream")
async def chat_stream(body: ChatBody):
    async def token_stream() -> AsyncGenerator[bytes, None]:
        # Produce the final answer, then stream it token-by-token style.
        answer = await llm_route_and_answer(body.message, body.user_id)
        for ch in answer:
            yield f"data: {ch}\n\n".encode("utf-8")
            await asyncio.sleep(0.01)
        yield b"data: [DONE]\n\n"

    return StreamingResponse(token_stream(), media_type="text/event-stream")

```

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host=HOST, port=PORT)
```

#### Dockerfile (optional)

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["python", "main.py"]
```

## 5) React Frontend — web-client/

package.json

```
{
  "name": "web-client",
  "private": true,
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "devDependencies": {
    "vite": "^5.2.0"
  }
}
```

vite.config.js

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    host: true,
    port: 5173
  }
})
```



```
}  
})
```

index.html

```
<!doctype html>  
<html>  
  <head>  
    <meta charset="UTF-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
    <title>Broco x Aakey – AI Agent</title>  
  </head>  
  <body>  
    <div id="root"></div>  
    <script type="module" src="/src/main.jsx"></script>  
  </body>  
</html>
```

src/styles.css

```
* { box-sizing: border-box; }  
body { font-family: ui-sans-serif, system-ui, -apple-system, Segoe UI,  
Roboto, Helvetica, Arial; margin: 0; background:#0b1020; color:#e7e9ee; }  
.container { max-width: 900px; margin: 0 auto; padding: 16px; }  
.card { background: #10162a; border: 1px solid #1d2642; border-radius: 16px;  
padding: 16px; box-shadow: 0 10px 25px rgba(0,0,0,0.25); }  
.messages { height: 65vh; overflow-y: auto; padding: 8px; display:flex; flex-  
direction:column; gap: 12px; }  
.msg { padding: 12px 14px; border-radius: 12px; max-width: 85%; white-space:  
pre-wrap; }  
.msg.user { background: #2b3a67; align-self:flex-end; }  
.msg.bot { background: #162040; align-self:flex-start; }  
.inputbar { display:flex; gap: 8px; margin-top: 12px; }  
input[type=text] { flex:1; padding: 12px; border-radius: 12px; border: 1px  
solid #2a355a; background:#0e1530; color:#e7e9ee; }  
button { padding: 12px 16px; border-radius: 12px; border: 0;  
background:#3b82f6; color:white; cursor:pointer; }  
button:disabled{ opacity: .6; cursor: not-allowed; }  
.small { opacity: .7; font-size: 12px; }
```

src/api.js

```
const BASE = import.meta.env.VITE_AGENT_BASE_URL || 'http://localhost:8000'  
  
export async function sendOnce(payload) {  
  const r = await fetch(`${BASE}/chat`, {
```

```

    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(payload)
  })
  return r.json()
}

export function streamChat(payload, onToken, onDone, onError) {
  const ctrl = new AbortController()
  fetch(`${BASE}/chat/stream`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(payload),
    signal: ctrl.signal
  }).then(async (res) => {
    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 })
      const parts = buffer.split("\n\n")
      buffer = parts.pop()
      for (const part of parts) {
        if (!part.startsWith('data:')) continue
        const data = part.replace(/^data:\s*/, '')
        if (data === '[DONE]') { onDone?(); return }
        onToken?.(data)
      }
    }
    onDone?()
  }).catch(err => onError?.(err))
  return () => ctrl.abort()
}

```

src/App.jsx

```

import { useEffect, useRef, useState } from 'react'
import './styles.css'
import { streamChat } from './api'

function Message({ role, content }) {
  return <div className={"msg " + (role === 'user' ? 'user' : 'bot')}>
    >{content}</div>
  }

export default function App() {
  const [messages, setMessages] = useState([

```

```



    { role: 'bot', content: 'Hey Aakey! I\'m Broco – your AI Agent. Ask me
for products or say your user id to fetch your profile. Try: "Show me
filament under 1400"' }
  ])
  const [input, setInput] = useState('')
  const [loading, setLoading] = useState(false)
  const scrollRef = useRef(null)

  useEffect(() => {
    if (scrollRef.current)
      scrollRef.current.scrollTop = scrollRef.current.scrollHeight
  }, [messages])

  const send = () => {
    if (!input.trim() || loading) return
    const userMsg = { role: 'user', content: input }
    setMessages(m => [...m, userMsg, { role: 'bot', content: '' }])
    setInput('')
    setLoading(true)

    const abort = streamChat({ message: userMsg.content, user_id: 101 },
(token) => {
      setMessages(m => {
        const copy = [...m]
        const last = copy[copy.length - 1]
        last.content += token
        return copy
      })
    },
    () => setLoading(false),
    () => setLoading(false)
  )

  return abort
}

return (
  <div className="container">
    <h2>Broco  Aakey  AI Agent</h2>
    <div className="small">Streaming with typing effect. Backend tools:
Node API + LLM routing.</div>
    <div ref={scrollRef} className="messages card">
      {messages.map((m, i) => <Message key={i} role={m.role}
content={m.content} />)}
    </div>
    <div className="inputbar">
      <input type="text" value={input} onChange={e =>
setInput(e.target.value)} placeholder="Type a task..." onKeyDown={e => e.key
=== 'Enter' ? send() : null} />
      <button onClick={send} disabled={loading}>Send</button>
    </div>
  </div>
)

```

```
    </div>
  )
}
```

src/main.jsx

```
import React from 'react'
import { createRoot } from 'react-dom/client'
import App from './App'

createRoot(document.getElementById('root')).render(<App />)
```

### Dockerfile (optional)

```
FROM node:20-alpine as base
WORKDIR /app
COPY package.json package-lock.json* ./
RUN npm ci || npm i
COPY . .
EXPOSE 5173
CMD ["npm", "run", "dev", "--", "--host"]
```

## 6) Run Locally (without Docker)

### Terminal 1 — Node API

```
cd node-api
npm i
npm start # http://localhost:4000
```

### Terminal 2 — Python Agent

```
cd py-agent
python -m venv .venv && source .venv/bin/activate #
Windows: .venv\Scripts\activate
pip install -r requirements.txt
export OPENAI_API_KEY=sk-... # optional; without it, agent uses rule-based
fallback
export NODE_API_BASE_URL=http://localhost:4000
python main.py # http://localhost:8000
```

### Terminal 3 — Web Client

```
cd web-client
npm i
npm run dev # http://localhost:5173
```

Open the web app and chat. You'll see **typing-style streaming** as the agent responds.

---

## 7) How it Works (quick)

1. React sends your prompt to `POST /chat/stream` on the Python agent.
  2. The AI Agent uses **OpenAI tool calling** (if key provided) to pick tools like `get_products` / `get_user` and calls the **Node API**.
  3. It summarizes results and **streams characters** back via **SSE**, which the UI renders as a typing effect.
- 

## 8) Extending the Agent

- Add more Node endpoints: orders, inventory, analytics.
  - In `TOOLS` (Python), add new function schemas (names must match your tool functions) and implement their HTTP calls in `call_node_api`.
  - Switch SSE granularity to words/tokens, or stream directly from the LLM API when desired.
  - Add auth (JWT) between web → agent, and agent → Node API.
- 

## 9) Notes

- If no `OPENAI_API_KEY` is provided, the agent still works using a simple rules engine, so you can demo end-to-end streaming immediately.
  - For production, consider a message store, retries, timeouts, observability (OpenTelemetry), and guardrails for tool calls.
- 

**You're ready, Aakey.** Ship it, then we can iterate features like multi-turn memory, RAG, and function/tool plugins next!