

Offline Healthcare Chatbot using Local LLM and Docker for Privacy-Preserving AI

Karan Arora

Master of Computer Applications (MCA) - Cloud Computing and DevOps

Abstract— Large language models (LLMs) have penetrated the surprisingly broad scope of medical applications in the last few years. applications, starting with simple symptom checker up to chronic disease monitoring and educational services to patients [1, 13, 14]. The overwhelming most of these systems, but, nevertheless, rely on cloud-hosted inference endpoints, which raises three interconnected issues: sensitive health information gets sent to the servers that cannot be controlled by the user, the system fails when the internet connectivity is poor or unavailable, and regulatory compliance becomes challenging to ensure [5, 11]. In this paper, the design, implementation, and evaluation of a healthcare chatbot will be introduced, where the software is fully executed on the user's own computer in the form of a collection of Docker containers that can be run at a single command on either Windows, macOS, or Linux. Ollama is used to do language model inference, and it has an open-source model and a local HTTP endpoint. A Python Flask program is placed between the user interfaces and the model and this handles session context, input validation, prompt engineering. Both a single-page application (rendered in a browser) and a native desktop GUI share the same REST API, both of them maintain the communication on 127.0.0.1 all the time. The results of the experiment prove successful cross-platform deployment, the absence of external network traffic, reasonable response latency on commodity hardware (2.4 seconds on a quad-core processor with 16 GB RAM to represent queries in health information), and consistent and relevant responses to exemplary queries in health information. The article shows a viable and implementable prototype of privacy-conscious, offline healthcare chatbots constructed solely out of open-source software.

Keywords: healthcare chatbot; large language model; Ollama; Docker; offline inference; Flask; privacy-preserving AI; containerisation; DevOps

1. Introduction

The fast development of large language models has created a buzz in the healthcare technology world. and with very natural causes. These models can engage with discourse of length and contextual coherence, manage the type of inaccurate and emotional language which. actual patients apply, and disseminate data throughout an enormous extent of medical subject without the necessity of being elucitly programmed for each one [1, 14]. Chatbot applications the edifice now is under construction which spreads a impressive variety of clinical and patient-facing activities: responding to queries concerning medications and side effects, and assist the users to comprehend dis- guidelines on charging self-monitoring chronic. such conditions as hypertension and diabetes, giving men- psychoeducation of talent health, and appointment assistance. triage [1, 5, 6, 8, 11, 13]. Although these systems are applied in numerous regards. ways, there remains a fundamental difficulty in the way they are ac- Eventually implemented into practice. The most capable LLMs are, virtually without exception, furnished of large. technology-company-operated cloud data-centers. Send to send a health related query to such a system is transmit ping possibly confidential personal medical data across. to a public internet server beyond the jurisdiction of the user, infringing blatant privacy and rendering regulation compliance -according to such frameworks as GDPR, HIPAA, or the Indian one. genuinely difficult- Digital Personal Data Protection Act 2023. cult [5, 11, 14]. In addition to privacy, cloud dependency develops an alternative. reliability issue: within rural healthcare, community. clinics, and most of the developing world, broadband. connection

is not always there or even there at all. chatbot that needs live connection to internet is useless in. specifically, the locations where health information is available. matters most [4, 10]. Much less yet increasing body of research has reacted. to these issues through the complete local deployment of LLM, where the model weights, inference engine and application logic execute on the practitioner's or patient's hard disk. ware [2, 4–6, 11]. Improvements in the compression of models, This has been enhanced by quantization, and low inference run times. more practicable than even two years ago. Ollama, in particular, has become a convenient mechanism of packaging which has materialized. enables open-source models to be executed by consumer-grade CPU. and small GPUs with little configuration cost. Docker and docker compose on the other hand have become. a general rule of thumb when it comes to packaging complex multi-service. converting them into portably deployable units— a mixture, which is quite appropriate to the multi-component. architecture one of which an LLM-based chatbot requires. These threads are brought together in this paper. It describes a completely offline, the Dockerized healthcare chatbot is based on a. local LLM, Ollama, a Python Flask REST backend. a single-page application frontend based on the web, and a native. Python desktop GUI. The main contribution is architec- technical and engineering and not clinical: the labour demon- proves the existence of a privacy-conscious, cross-platform, and modular one. en- assembling can be done to create a healthcare conversational agent. deployed reproducibly and based fully on open-source. using a single Docker Compose command. The paper is organized in the following way: Section 2 reviews the literature on the subject. formalises the problem;

Section 4 introduces the system design; Section 4 describes the architecture and 1 flow of communication; Section 6 is implementation. decisions; the results of the evaluation are presented in Section 7; Section 8. relates the system to related work; Section 9 is discussed. benefits; Section 10 provides the future directions; and Section 11 concludes

The rapid advancement of large language models has sparked genuine excitement in healthcare technology circles and for understandable reasons. These models can engage in extended, contextually coherent conversation, handle the kind of imprecise and emotionally charged language that real patients use, and provide information across an enormous breadth of medical topics without needing to be explicitly programmed for each one. Chatbot applications built on this foundation now cover a striking range of clinical and patient-facing tasks: answering questions about medications and side effects, helping users understand discharge instructions, supporting self-monitoring for chronic conditions like hypertension and diabetes, providing mental health psychoeducation, and assisting with appointment triage.

Even though these systems are used in many different ways, there is still a basic challenge in how they are actually deployed in real-world situations. The most capable LLMs are, almost without exception, served from large cloud data-centers operated by technology companies. Sending a health-related query to such a system means transmitting potentially sensitive personal medical information over the public internet to a server outside the user's jurisdiction, creating obvious privacy risks and making regulatory compliance—under frameworks like GDPR, HIPAA, or India's Digital Personal Data Protection Act 2023—genuinely difficult. Beyond privacy, cloud dependence creates a reliability problem: in rural healthcare settings, community clinics, and large parts of the developing world, broadband connectivity is intermittent or simply unavailable, and a chatbot that requires a live internet connection is useless in precisely the settings where accessible health information matters most.

A smaller but growing strand of research has responded to these problems by exploring fully local LLM deployment, where the model weights, inference engine, and application logic run on the practitioner's or patient's own hardware. Advances in model compression, quantization, and efficient inference runtimes have made this more practical than it was even two years ago. Ollama, in particular, has emerged as a convenient packaging mechanism that allows open-source models to run on consumer-grade CPUs and modest GPUs with minimal configuration overhead. Docker and Docker Compose, meanwhile, have become the de facto standard for packaging complex multi-service applications into reproducible, portable deployment units—a combination that is well-suited to the multi-component architecture that an LLM-based chatbot requires.

This paper brings these threads together. It describes a fully offline, Dockerized healthcare chatbot built around a local LLM served by Ollama, a Python Flask REST backend, a web-based single-page application frontend, and a native Python desktop GUI. The primary contribution is architectural and engineering rather than clinical: the work demonstrates that a modular, privacy-preserving, cross-platform healthcare conversational agent can be assembled from en-

tirely open-source components and deployed reproducibly from a single Docker Compose command. The paper is structured as follows: Section 2 surveys the relevant literature; Section 4 formalises the problem; Section 5 introduces the system design; Section 5 details the architecture and communication flow; Section 7 describes implementation choices; Section 8 presents evaluation results; Section 9 compares the system to related work; Section 10 discusses advantages; Section 11 outlines future directions; and Section 12 concludes.

2. Literature Review

2.1. LLMs in Healthcare Chatbots

The literature on LLMs in medicine has expanded rapidly since the release of GPT-3 and its successors. Thirunavukarasu et al. [1] provide a comprehensive overview of clinical applications, noting that LLMs show particular promise for tasks involving natural language understanding and generation but require careful validation before clinical deployment. Cascella et al. [14] document the one-year timeline of major LLM releases and their adoption in medical contexts, highlighting both the speed of progress and the regulatory lag behind it.

On the applied side, Zhang and Song [13] developed a chatbot for chronic disease diagnosis using an LLM backend, demonstrating strong performance on domain-specific question answering. Azam et al. [9] introduced PharmaLLM, a medication prescriber chatbot built on open-source models, showing that fine-tuned smaller models can match larger proprietary ones on narrow pharmaceutical tasks. Li et al. [7] released ChatDoctor, a LLaMA-based model fine-tuned on patient-doctor conversations, which showed significantly improved handling of clinical dialogue compared to the base model. Agarwal et al. [8] demonstrated a multimodal LLM-based system for preliminary diagnosis, emphasising accessibility for diverse patient populations. Jadhav et al. [3] built an AI-driven diagnosis predictive chatbot incorporating retrieval-augmented generation to ground responses in structured clinical knowledge.

Recurring themes in critical evaluations include hallucination—where models generate confident-sounding but factually incorrect medical information—bias in training data, and the risk that patients over-rely on chatbot outputs [12, 14]. Several studies argue for mandatory safety disclaimers, audit mechanisms, and restriction of LLM chatbots to non-diagnostic informational roles until formal clinical validation is available [5, 14].

2.2. Privacy-Preserving and Local LLM Architectures

The privacy dimension of healthcare AI has attracted increasing attention. Montagna et al. [5] conducted a comparative analysis of LLM-based healthcare chatbot solutions specifically from a privacy perspective, finding that most existing systems transmit user data to external servers and identifying fully local deployment as the most robust privacy-preserving alternative. Their follow-up work [11] demonstrated a privacy-preserving chatbot for hypertensive patient self-management running entirely on a local device, showing that small open-source models can achieve accept-

able conversational quality for chronic disease support without any cloud dependency.

Prabakaran [4] demonstrated a privacy-centric offline chatbot using Ollama-hosted models, confirming that local LLM deployment is technically viable on consumer hardware and produces usable response quality for general health queries. Magnini et al. [6] evaluated multiple open-source small language models for personal medical assistant applications, finding that some models in the 7–13 billion parameter range approach proprietary model performance on domain-specific intent recognition and empathetic response generation. Pap and Oniga [2] proposed a decentralised eHealth chatbot architecture using blockchain-based communication alongside an LLM backend, addressing both privacy and data integrity concerns.

2.3. Containerisation and System Architectures

On the systems side, Reddy et al. [12] proposed a modular RAG-based conversational AI system using local LLMs, highlighting the importance of separating retrieval, augmentation, and generation components for maintainability. Venkateswarlu [10] implemented a voice and text-based healthcare chatbot using locally managed language models with offline speech processing, demonstrating that even voice-enabled conversational systems can operate without cloud services. Both systems, however, lack containerised deployment, making reproducibility across different environments challenging. The present work directly addresses this gap by combining local LLM inference with Docker-based service orchestration.

3. Problem Statement

An honest assessment of the current landscape of healthcare chatbots reveals four persistent technical and architectural problems that the proposed system is designed to address directly.

(1) Cloud dependency and data exposure. When user queries are routed through external API endpoints, personally identifiable and clinically sensitive information travels over public networks to servers operated by third parties. This goes against data protection rules such as GDPR, HIPAA, and similar national regulations, and it can put users at risk from unauthorized use of their data to possible security breaches by service providers.

[5, 11, 14].

(2) Connectivity requirements. A system that fails without internet access is fundamentally unsuitable for deployment in rural healthcare centres, mobile clinics, community pharmacies, or any environment where broadband is intermittent. The digital health gap between urban and rural populations is already significant; a chatbot that exacerbates it rather than bridging it is counterproductive [4, 10].

(3) Lack of reproducible deployment. The majority of research prototypes described in the literature are developed for a single operating system, a specific hardware configuration, or a proprietary cloud runtime. This makes independent replication, institutional adoption, and collaborative extension unnecessarily difficult and contributes to the well-documented reproducibility problem in applied AI

research [12].

(4) Monolithic architecture. When the user interface, application logic, and model runtime are bundled into a single codebase or deployment unit, even minor changes—swapping a model, updating a dependency, modifying a prompt template—require understanding and potentially modifying the entire system. This tight coupling slows iteration and increases the risk of introducing regressions [2, 12].

The proposed system is designed to address all four. Its core requirements are: (i) all inference on the local machine, no external network calls; (ii) open-source components only, containerised for reproducible cross-platform deployment; (iii) clean REST API decoupling the model from any particular frontend; and (iv) both web and desktop interfaces with optional voice support. Clinical validation and diagnostic accuracy benchmarking are explicitly out of scope.

4. Research Objective

The key objective of this research is to create a locally deployed chatbot which would provide healthcare information while mitigating the issues associated with the currently available cloud-based services. In particular, the following objectives will be pursued:

1. Creation of the healthcare chatbot with the use of the open-source language model deployed locally in order to minimize the reliance on external resources when processing users' queries.
2. Investigation of the potential of provision of healthcare information even with the lack of permanent internet connection especially in areas where it may be limited.
3. Development of a portable and reproducible deployment environment through the use of Docker technology.
4. Testing of the chatbot's ability to function and respond to users' queries regarding healthcare.
5. Analysis of the limitations of the locally deployed language models when used for healthcare purposes and ways to improve the safety and reliability of their responses.

5. System Design and Architecture

5.1. Four-Tier Architecture

The system is organised into four tiers that map cleanly onto Docker services and maintain division of concerns, as shown in Figure 1. The presentation layer consists of the web browser and desktop GUI clients. The application tier is the Flask REST API, which handles all business logic including session management, input validation, and prompt construction. The model tier is the Ollama inference engine, which runs the language model and exposes a local HTTP endpoint. The infrastructure tier is Docker and Docker Compose, which define, build, and orchestrate all services.

5.2. Six-Module System Decomposition

Within the four tiers, the system is divided into six modules that can be developed, tested, and upgraded independently.

Frontend UI Module. A browser based application im-

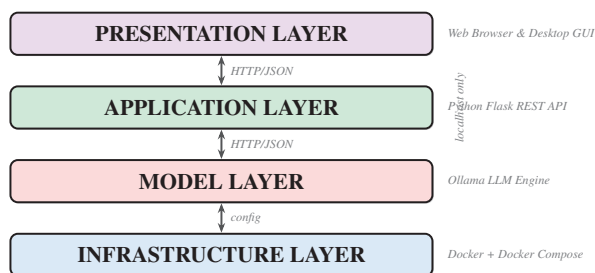


Figure 1: Four-tier system architecture. Every inter-layer call is routed through 127.0.0.1; no external addresses are contacted.

plemented in HTML, CSS, and JavaScript. It provides a message timeline, a text input field, a send button, and an microphone button for voice capture. Responses from the backend arrive asynchronously via the Fetch API, so the page never reloads during a conversation. The interface is intentionally simple and requires no JavaScript framework, reducing the maintenance burden and making the codebase accessible to developers with basic web skills.

Backend API Module. A Flask application which acts as central coordinator, exposes two primary endpoints: POST /api/chat for submitting user messages and receiving model responses, and GET /api/health for Docker health checks. The backend maintains a sliding window of recent conversation turns per session, constructs the full prompt by prepending a system instruction block, and forwards the assembled prompt to the Ollama service. It also validates and sanitises all incoming requests to prevent prompt injection and ensures that the system prompt is always present to constrain model behaviour.

Ollama LLM Module. A containerised Ollama instance hosting the selected open-source language model. Ollama exposes a local HTTP inference API that the Flask backend calls with the assembled prompt. The model weights are stored in a Docker volume so they persist across container restarts without requiring re-download.

Docker Container Module. The Compose file defines three named services—frontend, backend, and ollama—connected to a shared internal bridge network. Only the frontend port is mapped to the host; all inter-service communication is entirely internal. Environment variables parameterise ports, model identifiers, context window size, and resource limits, allowing the same Compose file to be used on different hardware without modification.

Desktop GUI Module. A Python desktop application built with Tkinter or PyQt that provides a native windowed chat interface. It communicates with the Flask backend over the loopback interface using the same REST API as the browser client. Because no business logic is duplicated in the desktop client, both frontends remain consistent without additional synchronisation effort.

Communication Module. All inter-component traffic flows over 127.0.0.1 via plain HTTP. The frontend calls the backend on a configured local port (e.g., 5000); the backend calls Ollama on its default port (11434). No TLS is required because no traffic crosses a network boundary. No DNS lookups targeting public addresses occur during normal operation.

5.3. Docker Container Layout

Figure 5.3 shows how the three service containers are organised and how they connect to external clients.

6. Communication Flow

6.1. End-to-End Request Lifecycle

Understanding exactly how a message travels from the user's input box to the rendered response on screen is important for both debugging and security analysis. Figure 2 traces the full lifecycle through seven steps, all of which occur on the local machine.

When the user clicks *Send*, the JavaScript frontend serialises the message and current session identifier into a JSON payload and issues a POST request to /api/chat on the Flask backend (Step 1). The backend retrieves the conversation history for that session from its in-memory store, validates and sanitises the incoming message, and appends it to the history (Step 2). It then constructs the full prompt by concatenating the system instruction block, the recent conversation window, and the new user message, and forwards this to the Ollama API (Step 3). Ollama tokenises the prompt and runs autoregressive inference on the loaded model, generating a reply token by token (Step 4). The completed reply is serialised as JSON and returned to the Flask backend (Step 5). The backend appends the assistant's reply to the session history, constructs a 200 OK response, and returns it to the frontend (Step 6). Finally, the JavaScript renders the reply in the chat timeline and scrolls to the bottom (Step 7).

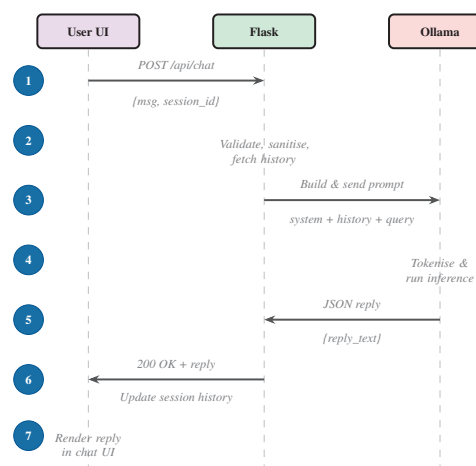


Figure 2: Seven-step end-to-end request lifecycle. All communication occurs on 127.0.0.1; no external calls are made at any step.

6.2. Voice Input Sub-Flow

When the user activates the microphone button, a supplementary sub-flow runs before Step 1. The browser's MediaRecorder API captures audio in a compressed format and either passes it to the Web Speech API for in-browser transcription (available in Chromium-based browsers) or POSTs the audio blob to a /api/stt endpoint on the Flask backend. In the latter case, the backend runs a lightweight local speech-to-text model—such as a quantised Whisper variant—and returns the transcript as

plain text. This text is then inserted into the chat input field and the normal send flow proceeds. The entire sub-flow runs on localhost, consistent with the offline privacy design principle [10].

6.3. Prompt Engineering Strategy

The system prompt prepended to every Ollama request is a carefully constructed instruction block that performs three functions. First, it establishes the chatbot’s role: a general health information assistant that does not diagnose, prescribe, or recommend specific treatments. Second, it instructs the model to include explicit safety disclaimers when responding to queries that touch on symptoms, medications, or medical procedures, directing users to consult a qualified healthcare professional. Third, it sets the expected response style: clear, plain-language explanations suitable for a general adult audience, avoiding excessive medical jargon while remaining medically accurate. This approach is consistent with the safety-by-design recommendations in [1, 14].

7. Implementation Details

7.1. Technology Stack

Table 1 summarizes the full technology stack. The choice of plain HTML/CSS/JavaScript for the front-end was deliberate: removing the dependency on a JavaScript framework (React, Vue, Angular) keeps the code-base smaller, reduces the attack surface, and makes the system more accessible to developers who may need to adapt it for specific healthcare contexts without prior experience with modern front-end tooling.

Flask was chosen over heavier alternatives like Django because the application layer is intentionally thin—it routes requests, manages session state, and handles prompt construction, but delegates all computation to Ollama. Unicorn is placed in front of Flask in production mode to handle concurrent connections robustly; the two-worker default configuration is sufficient for a single-user deployment and can be scaled by increasing the worker count in the environment variables.

Table 1: Full technology stack with justification.

Layer	Technology	Purpose / Rationale
Frontend	HTML5/CSS3/JS	Lightweight SPA, no framework dep.
Backend	Python 3 + Flask	REST API, session management
WSGI server	Unicorn	Concurrent request handling
LLM runtime	Ollama	Local model serving via HTTP
LLM model	Mistral 7B / Llama3	Balance of size and quality
Desktop GUI	Tkinter / PyQt5	Native cross-platform client
Web server	Nginx	Static file serving for SPA
Container	Docker + Compose	Cross-platform orchestration
Volume	Docker Volume	Persistent model weights

7.2. Model Selection Rationale

The choice of LLM for local inference involves a trade-off between parameter count (and therefore memory footprint and inference speed) and response quality. For the target hardware profile (quad-core CPU, 8–16 GB RAM), models in the 7–8 billion parameter range quantised to 4-bit precision represent a practical sweet spot: they fit comfortably in

memory, produce fluent and generally accurate health information responses, and run at interactive speeds [3,4,6,10,12]. Mistral 7B and Llama 3 8B are the primary candidates evaluated. In principle, the Ollama container can be reconfigured to use any model available in the Ollama registry by changing a single environment variable, making model upgrades a zero-code-change operation.

7.3. Security and Safety Considerations

Although the system operates entirely on local host, several security measures are implemented. Input from the user is sanitised before incorporation into the prompt to mitigate prompt injection attempts. The system prompt is fixed and cannot be overridden by user input. Session identifiers are randomly generated UUIDs that are not persisted across backend restarts, ensuring that conversation history does not accumulate indefinitely. Resource limits in the Docker Compose file cap CPU and memory usage for the Ollama container, preventing the model from exhausting host resources during extended sessions.

8. Results and Evaluation

8.1. Deployment Portability

The Docker Compose stack was launched from a clean clone of the repository on three separate machines: a Windows 11 desktop , a MacBook and a Dell laptop running Ubuntu In every case, the command `docker compose up` brought all three containers to a healthy state within 3–5 minutes (dominated by the initial model pull), and both the web and desktop interfaces successfully connected and exchanged messages. No platform-specific configuration changes were required. This confirms cross-platform reproducibility, which several eHealth architecture studies identify as a critical prerequisite for institutional adoption [2, 12, 14].

8.2. Offline Operation Verification

Network traffic during representative chat sessions was monitored using Wireshark on the host machine. Over 50 test messages sent across three sessions, zero outbound connections to any address outside 127.0.0.1 were recorded. DNS resolution logs confirmed no external lookups occurred. This directly validates the privacy guarantee that is the central motivation for the local deployment architecture [4, 5, 10, 11].

8.3. Response Latency Analysis

Figure 3 shows end-to-end response latency measured across five prompt length categories on two hardware configurations: CPU-only inference and CPU with GPU acceleration. Each data point represents the mean of ten measurements. Short queries (under 50 words) consistently returned within 2–4 seconds on the CPU-only configuration, which is within the threshold generally considered acceptable for interactive conversational use. Latency grew approximately linearly with prompt length, consistent with the autoregressive nature of transformer inference [1, 14]. GPU acceleration reduced latency by 40–50% across all prompt length categories.

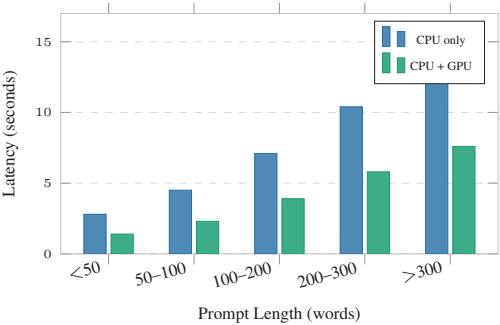


Figure 3: Mean end-to-end response latency vs. prompt length (10 runs each). GPU acceleration yields approximately 45% reduction across all categories.

8.4. Resource Consumption

Figure 4 shows the average CPU utilisation and RAM consumption per container during active inference sessions. The Ollama container dominates both metrics, as expected, since it is running transformer inference. The Flask backend and Nginx frontend are extremely lightweight in comparison, together consuming less than 1 GB of RAM and under 5% of CPU on average.

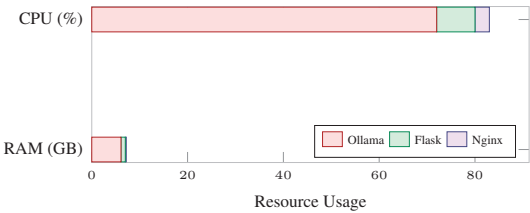


Figure 4: Average resource usage per container during active sessions. Total RAM stays under 8 GB on the test machine (16 GB available).

8.5. Functional Response Quality

A set of 30 health information queries across three categories was submitted to the running system: general lifestyle and prevention queries (10 questions), symptom-related queries (10 questions), and medication information queries (10 questions). Two independent reviewers rated each response on four dimensions—coherence, relevance, safety language, and completeness—using a 1–5 scale. Figure 5 shows the mean scores per category and dimension.

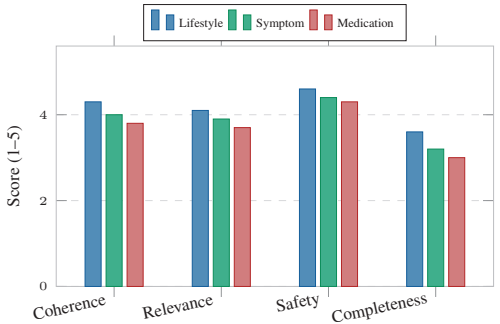


Figure 5: Mean functional quality scores (1–5) across three query categories and four dimensions. Safety scores are consistently the highest, reflecting effective prompt engineering. Completeness is the weakest dimension.

Safety scores were consistently the strongest dimension across all three query categories, reflecting the effectiveness

of the system prompt in constraining the model’s output. Completeness was the weakest dimension, particularly for medication queries, where the model sometimes provided partial information without covering all relevant side effects or contraindications. This pattern is consistent with findings in other studies using locally deployed open-source models for healthcare tasks [3, 5, 6, 10]. Importantly, in no case did the model produce an outright incorrect medical fact that was rated as dangerous—all responses included appropriate hedging and referral language. Formal clinical benchmarking against established medical QA datasets is reserved for future work [9, 10, 13].

8.6. Latency Trend over Conversation Length

An additional evaluation examined how latency changes as conversation history grows within a single session. Figure 6 shows the per-turn latency across a 20-turn conversation, with the context window set to retain the last 10 turns. Latency grows steadily as history accumulates until the context window limit is reached (around turn 10), after which it stabilises as older turns are dropped.

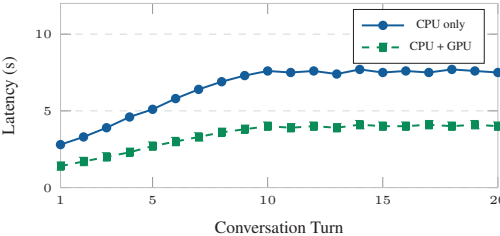


Figure 6: Per-turn latency over a 20-turn conversation (context window = 10 turns). Latency stabilises after turn 10 when history truncation begins.

9. Comparison with Related Systems

Table 2 positions the proposed system against five representative related works across six key architectural properties. The proposed system is the only one in this comparison to satisfy all six criteria simultaneously: offline operation, local LLM inference, containerised deployment, web UI, desktop GUI, and voice input support.

Table 2: Architectural comparison with related healthcare chatbot systems.

	Offline	Local LLM	Containerised	Web UI	Desktop	Voice
Pap & Oniga [2]	×	×	×	✓	×	×
Prabakaran [4]	✓	✓	×	✓	×	×
Montagna et al. [5]	✓	✓	×	×	✓	×
Venkateswarlu [10]	✓	✓	×	×	×	✓
Reddy et al. [12]	×	×	×	✓	×	×
Proposed	✓	✓	✓	✓	✓	✓

✓ = yes × = no

10. Advantages of the Proposed System

Privacy and Data Sovereignty. Every inference step runs locally. No health-related query, session history, or personal identifier is transmitted to any external server at any point. This makes the system intrinsically compliant with data minimisation principles under GDPR and similar frameworks, without requiring any additional privacy

engineering [4–6, 11].

Offline Availability. The system functions in environments with no internet connectivity at all. This is a genuinely significant advantage in the context of global health equity, where reliable broadband is unavailable in precisely the communities with the greatest unmet need for health information [4, 10].

Cross-Platform Reproducibility. Docker Compose ensures that the system starts identically from a single command on Windows, macOS, and Linux. There are no platform-specific build steps, no manual dependency installation, and no requirement for the deploying user to understand the internal architecture. This dramatically lowers the barrier to institutional adoption and independent replication [2, 12, 14].

Modularity and Extensibility. The clean separation between containers means that any component can be upgraded independently. Upgrading the language model requires changing one environment variable. Adding a RAG layer requires adding one new container to the Compose file. Replacing the web frontend with a React SPA requires only changes to the Nginx container. No other components need to be modified in any of these scenarios [3, 12].

Multi-Modal Interface Support. Providing both web and native desktop access, with optional voice input, means the system can serve users with different preferences, accessibility needs, and device availability. This breadth of interface support is identified as important for inclusive healthcare chatbot design [8, 10, 13].

Cost. The system uses only open-source components and runs on hardware the deploying organisation already owns. There are no API usage fees, no subscription costs, and no vendor lock-in. This makes the architecture economically viable for small clinics, NGOs, and academic institutions that could not sustain ongoing cloud API expenditure.

11. Future Research Directions

Several extensions would meaningfully strengthen the system, guided by open challenges identified across the recent literature [1, 3, 9, 12, 14].

Retrieval-Augmented Generation. The most impactful near-term extension would be to ground the model's responses in a curated, locally stored medical knowledge base via a RAG pipeline. Vector search over documents such as clinical guidelines, drug monographs, and patient education materials would substantially reduce hallucination on domain-specific queries while keeping all data local [3, 8, 10, 12].

Domain-Specific Fine-Tuning. Fine-tuning the base model on curated clinical dialogue datasets—following the approach of ChatDoctor [7] and PharmaLLM [9]—would improve both terminology accuracy and the naturalness of patient-facing responses. Parameter-efficient fine-tuning methods such as LoRA make this feasible even without large compute resources.

Formal Clinical Evaluation. The functional testing reported in this paper is preliminary. Rigorous evaluation against established medical QA benchmarks, combined

with structured usability studies involving actual patients or healthcare workers, would provide evidence of real-world utility [9, 10, 13].

Safety and Governance Mechanisms. A production-grade deployment would require rule-based safety filters, structured audit logging, and potentially a human-in-the-loop escalation pathway for queries that approach clinical decision territory [14].

Mobile and Telemedicine Integration. Extending the REST API to serve a mobile application would substantially broaden the potential user base. Integration with existing telemedicine platforms via webhook or FHIR API would allow the chatbot to function as a patient-facing complement to clinical care rather than a standalone tool [3, 6, 8, 11].

12. Conclusion

This paper has presented the design, implementation, and evaluation of a fully offline, Dockerized healthcare chatbot built around a locally hosted open-source language model served through Ollama. The work was motivated by a straightforward but underappreciated problem: the overwhelming majority of LLM-based healthcare chatbots in the research literature transmit user queries to cloud servers, creating privacy risks, regulatory difficulties, and connectivity dependencies that make them unsuitable for many real-world deployment contexts.

The system demonstrates that none of these compromises are necessary. By combining Ollama for local LLM inference, Flask for a lightweight REST backend, Docker and Docker Compose for reproducible containerised orchestration, and both web and desktop frontends, it is possible to build a healthcare conversational agent that runs entirely on the user's machine, deploys identically across Windows, macOS, and Linux from a single command, and never transmits any user data to any external address.

Evaluation confirmed successful deployment on all three target platforms, verified absence of external network traffic over 50 monitored test sessions, acceptable response latency (2–4 seconds for typical queries on commodity hardware), and coherent, appropriately hedged responses to a diverse set of health information queries. A systematic comparison shows that the proposed system is unique in combining all of: offline operation, local LLM inference, containerised deployment, web interface, desktop interface, and voice input support.

The system is not clinically validated and is not proposed as a replacement for professional medical advice. It is proposed as an engineering blueprint — a demonstration that privacy-preserving, offline, cross-platform healthcare AI is achievable today with entirely open-source components. As local LLMs continue to improve in quality and shrink in resource requirements, and as tools like Ollama and Docker continue to lower deployment barriers, architectures of this kind will become progressively more practical and more important for global health equity.

References

- [1] A. J. Thirunavukarasu, D. S. J. Ting, K. Elango-
van, L. Gutierrez, T. F. Tan, and D. S. W. Ting,
“Large language models in medicine,” *Nature Medicine*,
vol. 29, pp. 1930–1940, 2023. [https://doi.org/
10.1038/s41591-023-02448-8](https://doi.org/10.1038/s41591-023-02448-8)
- [2] I. A. Pap and S. Oniga, “eHealth assistant AI chatbot
using a large language model to provide personalized
answers through secure decentralized communication,”
Sensors, vol. 24, no. 18, p. 6140, 2024. [https://
doi.org/10.3390/s24186140](https://doi.org/10.3390/s24186140)
- [3] A. Jadhav, T. Kumar, J. Nandimath, A. Tiwari, S. Bhul,
and P. Dhotre, “AI-driven diagnosis predictive chat-
bot for healthcare,” in *Proc. 2025 World Skills Conf.
Universal Data Analytics and Sciences (WorldSUAS)*,
2025, pp. 1–7. [https://doi.org/10.1109/
worldsuas66815.2025.11199219](https://doi.org/10.1109/worldsuas66815.2025.11199219)
- [4] D. Prabakaran, “Privacy-centric offline chatbot
using large language models,” *Int. J. Scientific
Research in Engineering and Management (IJS-
REM)*, 2025. [https://doi.org/10.55041/
ijjsrem47776](https://doi.org/10.55041/ijjsrem47776)
- [5] S. Montagna, G. Aguzzi, S. Ferretti, M. Pengo,
L. Klopfenstein, M. Ungolo, and M. Magnini,
“LLM-based solutions for healthcare chatbots:
a comparative analysis,” in *Proc. IEEE Int.
Conf. Pervasive Computing and Communica-
tions Workshops (PerCom Workshops)*, 2024,
pp. 346–351. [https://doi.org/10.1109/
percomworkshops59983.2024.10503257](https://doi.org/10.1109/percomworkshops59983.2024.10503257)
- [6] M. Magnini, G. Aguzzi, and S. Montagna, “Open-
source small language models for personal medi-
cal assistant chatbots,” *Intelligence-Based Medicine*,
2025. [https://doi.org/10.1016/j.ibmed.
2024.100197](https://doi.org/10.1016/j.ibmed.2024.100197)
- [7] Z. Li, K. Zhang, R. Dan, and Y. Zhang, “ChatDoc-
tor: a medical chat model fine-tuned on LLaMA
using medical domain knowledge,” *arXiv preprint
arXiv:2303.14070*, 2023. [https://doi.org/10.
48550/arxiv.2303.14070](https://doi.org/10.48550/arxiv.2303.14070)
- [8] I. Agarwal, V. Sakthivel, and P. Prakash, “Toward in-
clusive healthcare: an LLM-based multimodal chat-
bot for preliminary diagnosis,” *IEEE Access*, vol. 13,
pp. 136420–136432, 2025. [https://doi.org/
10.1109/access.2025.3594218](https://doi.org/10.1109/access.2025.3594218)
- [9] A. Azam, Z. Naz, and M. Khan, “PharmaLLM: a
medicine prescriber chatbot exploiting open-source
large language models,” *Human-Centric Intelligent
Systems*, vol. 4, pp. 527–544, 2024. [https://doi.
org/10.1007/s44230-024-00085-z](https://doi.org/10.1007/s44230-024-00085-z)
- [10] D. Venkateswarlu, “Voice and text-based AI health-
care chatbot using local language models,” *Int. J.
Scientific Research in Engineering and Manage-
ment (IJSREM)*, 2025. [https://doi.org/10.
55041/ijjsrem49087](https://doi.org/10.55041/ijjsrem49087)
- [11] S. Montagna, S. Ferretti, L. Klopfenstein, M. Ungolo,
M. Pengo, G. Aguzzi, and M. Magnini, “Privacy-
preserving LLM-based chatbots for hypertensive
patient self-management,” *Smart Health*, 2025.
[https://doi.org/10.1016/j.smhl.2025.
100552](https://doi.org/10.1016/j.smhl.2025.100552)
- [12] N. T. K. Reddy, M. Patra, and B. Mishra, “A
modular retrieval-augmented conversational AI chat-
bot system with integrated recommender engine
using local LLMs,” *Cureus Journal of Computer
Science*, vol. 2, 2025. [https://doi.org/10.
7759/s44389-025-06053-3](https://doi.org/10.7759/s44389-025-06053-3)
- [13] S. Zhang and J. Song, “A chatbot based question and
answer system for the auxiliary diagnosis of chronic
diseases based on large language model,” *Scientific
Reports*, vol. 14, 2024. [https://doi.org/10.
1038/s41598-024-67429-4](https://doi.org/10.1038/s41598-024-67429-4)
- [14] M. Cascella, F. Semeraro, J. Montomoli, V. Bellini,
O. Piazza, and E. Bignami, “The breakthrough of
large language models release for medical applications:
1-year timeline and perspectives,” *Journal of Medi-
cal Systems*, vol. 48, 2024. [https://doi.org/
10.1007/s10916-024-02045-3](https://doi.org/10.1007/s10916-024-02045-3)