International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 13 Issue: 09 | Sep 2026 www.irjet.net p-ISSN: 2395-0072

A Privacy-Preserving, Multi-User Voice Assistant with Local LLM Reasoning, Persistent Memory, and Face-Assisted Authentication

Dupakuntla Durga Sri Krishna Deepika[1], G. Bhanu Prakash[2], Jasthi Sai Vignesh[3], Sri Lekha Pusuluru[4], SK. Mohammad Rafi[5]

[1,3,4,5] Student, Department of Artificial Intelligence and Data Science, Lingayas Institute of Management and Technology, Vijayawada, Andhra Pradesh, India
[2] Assistant Professor, Department of Artificial Intelligence and Data Science, Lingayas Institute of Management and Technology, Vijayawada, Andhra Pradesh, India

---

Abstract – *Most of the voice assistants people use every day run on cloud services. That keeps them dependent on an internet connection, ties them to paid subscriptions, and sends whatever the user says to servers the user does not control. In an earlier paper we built a Python desktop assistant that recognised spoken commands, mapped them to a fixed set of actions, and handed open-ended questions to a cloud language model. It worked, but it kept no memory between sessions, served only one user, understood only the exact phrases it was written for, and offered nothing beyond a desktop window. This paper describes a rebuilt version of that assistant. Rather than relying on keyword matching alone, it splits the work in two: predictable actions such as opening an application, taking a screenshot, or changing the volume are handled directly in code, while anything conversational is sent to a language model that runs locally through Ollama. The same local model also answers questions about uploaded images and summarises web-search results. Around this core we added separate user profiles with a PIN and an optional face sign-in, per-user memory and chat history kept in SQLite, a calendar that understands plain-English reminders, a local file search, and a browser-based interface. None of it needs a paid API key. Measured on an ordinary laptop with no graphics card, direct commands return in well under a millisecond while model replies take a few seconds, and the reminder parser, the PIN check, and memory recall were correct on every case we tested.*

Key Words: Conversational AI, Voice Assistant, Local Large Language Model, Human–Computer Interaction, Multi-User Authentication, Face Recognition, Natural Language Processing, Privacy-Preserving AI, Edge Inference

1. INTRODUCTION

Voice assistants have become a normal way to work with a computer without touching it. The ones most people know Siri, Alexa, Google Assistant do the heavy lifting in the cloud. That design has three well-known costs. It needs a network connection to work at all, it usually depends on paid services or dedicated hardware, and it sends the user's speech to a company's servers, where it may be logged and kept [6]. For anyone who wants an assistant that still works offline and keeps personal data on their own machine, those costs matter.

Our earlier project [1] was an assistant of exactly this kind. It waited for a wake phrase, turned the user's speech into text, and looked the text up in a table of commands to open applications, read out the time or weather, play media, and set reminders. General questions it could not answer on its own were forwarded to a cloud language model through the OpenAI API, and the reply was read back aloud. The system was fast and reliable, finishing most tasks in under 1.5 seconds, but it had two shortcomings we pointed out ourselves in that paper: it remembered nothing once a session ended, and its ability to hold a conversation depended completely on a paid cloud service. It was also written for a single person, with no idea of separate users.
This work takes that assistant apart and rebuilds it to remove those limits while keeping what already worked. The result runs its language and vision models locally, remembers each user separately, and is reached through a web browser instead of a single desktop window.

1.1 What the original system was missing

Looking back at [1], the gaps that mattered most were these:

- **It only understood exact phrases.** A command had to match one of the programmed keywords; a small change in wording and the assistant did nothing.
- **It forgot everything between sessions.** There was no lasting memory, so it could not use anything the user had told it earlier.
- **It leaned on a cloud model for conversation.** Open-ended replies came from the OpenAI API, which costs money and sends text off the machine.
- **It served one user.** There were no profiles, no sign-in, and no per-user settings or history.
- **It could not see or reason about images and** had no interface a second device could reach.

1.2 What this paper adds

1. **A two-part command router** that sends reliable, safety-sensitive actions straight to code and leaves everything else to a language model. This avoids asking a small local model to pick tools on its own, which they do poorly, while still allowing free conversation.
2. **A local, per-user data layer in SQLite** that holds accounts, remembered facts, chat history, calendar events, settings, and face descriptors in one place.
3. **Multi-user sign-in with a PIN and an optional face login,** something the original had no notion of. PINs are stored as salted PBKDF2-HMAC-SHA256 hashes, and face matching runs in the browser.
4. **A move from cloud to local inference.** Conversation and image understanding now run on the user's own machine through Ollama, so no text or image leaves it and no paid key is needed.
5. **A browser interface and several new features** that the original lacked: lasting per-user memory, image question answering, plain-English reminders, and a safe local file search.

2. RELATED WORK

Rule-based assistants, including our own earlier system [1], map recognised speech to actions with hand-written keyword rules. They are easy to follow and dependable for the commands they know, but they break the moment a user phrases something in a way the author did not anticipate. We keep a rule-based path for actions that must be predictable, and hand the rest to a language model.

Commercial cloud assistants recognise speech very accurately by running large models on remote servers, but studies of smart-speaker users report ongoing unease about how recorded audio is handled and stored, and show that many users are unclear on what happens to it [6]. This is a large part of why we chose to keep inference on the device. Running capable language models locally has only recently become practical. Open-weight models such as the LLaMA family [3], [4], built on the transformer architecture [2], can now run on ordinary hardware, and tools such as llama.cpp [9] and the Ollama server [8] make them easy to call through a simple local interface. We use Ollama to host both a text model for chat and a vision-language model for images.

For sign-in, we rely on face embeddings in the style of FaceNet [5], where two pictures of the same person map to nearby points and can be compared by distance. The face-api.js library [10] runs this pipeline inside the browser, so the raw camera image never leaves the client; only a 128-number descriptor is stored.

Finally, giving an assistant a lasting memory is known to make it noticeably more useful. Recent agents keep a store of facts and pull the relevant ones back into the model's context [7], in the same spirit as retrieval-augmented generation [11]. We use a lightweight version of this idea: facts the user asks us to remember are saved per user and added to the model's prompt on later turns. Notably, our earlier paper [1] listed exactly this persistent memory as missing future work, and it is one of the main gaps the present system closes.

3. PROPOSED SYSTEM AND METHODOLOGY

3.1 Overall design

The whole system runs on the user's own computer as a small client and server (Fig -1). The browser collects typed or spoken input speech is captured with the Web Speech API [13] and sends it to a Flask backend over a simple JSON interface. The backend checks that the user is signed in, then passes the request to a router. If the request is a known action, for example “open Teams”, “screenshot”, “volume up”, “remind me to …”, or “find file …”, a Python function carries it out and returns a fixed reply. Anything else is treated as conversation and sent to the local model through Ollama, with the answer streamed back to the browser a few words at a time. Everything that has to survive a restart accounts, PIN hashes, face descriptors, remembered facts, chat history, calendar events, and settings lives in a local SQLite file, so the assistant stays self-contained and needs no paid cloud service.

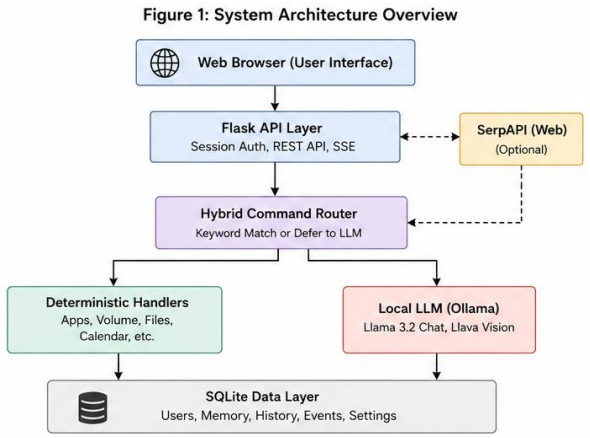

Fig -1: Overall system architecture. All processing and storage stay on the local machine.

3.2 Signing in

A profile is just a display name and a numeric PIN. The PIN is never written down as plain text. Instead the system generates a random 16-byte salt and stores a PBKDF2-HMAC-SHA256 hash of the PIN using 200,000 iterations [12] and checks it later with a constant-time comparison. A user can also enrol their face for quick sign-in. The face is turned into a 128-number descriptor in the browser using face-api.js [10]; at login the live descriptor is compared against

the stored ones by straight-line distance, and the closest match within a set threshold identifies the user. Because this happens in the browser, the actual face image is never uploaded. We treat face login as a convenience only: it has no liveness check and could be fooled by a photograph, so a PIN is always kept as the fallback and is required before any account change. Fig -2 traces the sign-in path, and Fig -3 shows the profile-creation screen.

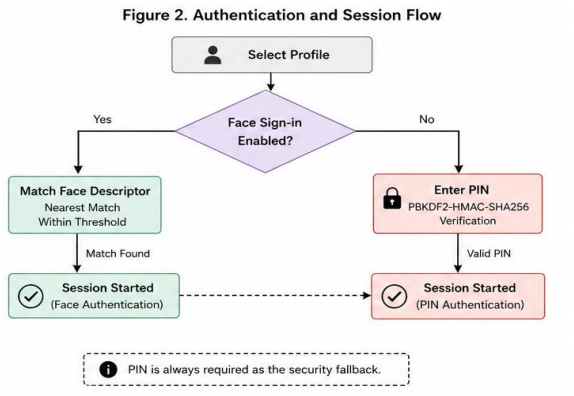

Fig -2: Authentication and session flow: profile selection, optional face match, and PIN fallback.

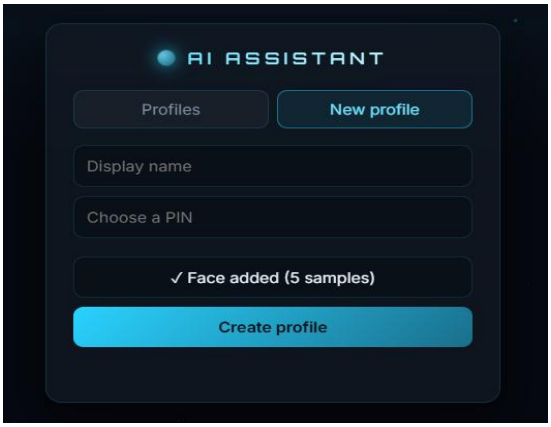

Fig -3: Creating a profile: display name, PIN, and optional face enrolment (five samples).

3.3 Deciding where a request goes

Before any model is involved, the router checks the request against a short list of known actions: open, close, or launch an app or website; take a screenshot; turn the volume up, down, or off; play, pause, or skip media; report the date or time; save or recall a fact; add or list calendar items; and search for a file. If one of these matches, the matching Python function runs and returns a fixed reply. Only when nothing matches the request go to the model, together with a short system prompt built from the user's profile and remembered facts. We split the work this way on purpose. Small local models are unreliable at choosing and calling tools by themselves, so tying real actions starting or killing a process, changing the volume to explicit rules keeps them predictable, and the model is left to do the open-ended reasoning it is actually good at. Fig -4 traces how a single request is decided.

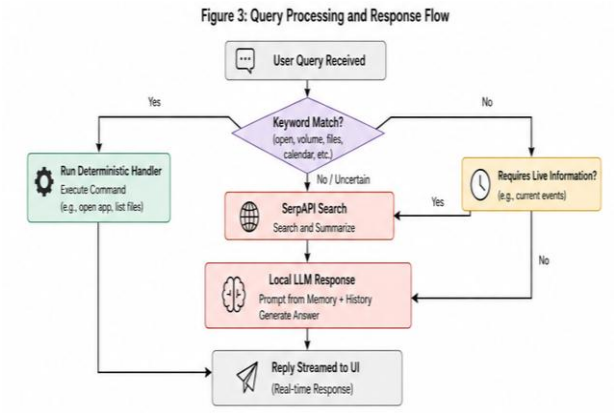

Fig -4: Query-processing decision flow: a request is matched against known actions first, and only unmatched requests reach the local model.

3.4 Memory and history

When a user says something like “remember that my exam is on Friday”, the fact is stored under that user in SQLite and added to the model's prompt on later turns, so the assistant carries context from one session to the next. This is the capability the earlier system did not have, where a “remember” note simply went into one shared file. Every turn of the conversation is also logged per user, and the most recent turns are replayed as context, so a multi-turn chat stays coherent. Users can view or clear their facts and history from the settings page.

3.5 Reminders, file search, and vision

The calendar understands ordinary requests. Phrases like “remind me to submit the report at 5 pm”, “schedule dentist tomorrow at 9:30 am”, or “remind me to stretch in 2 hours” are turned into scheduled events. The parser handles relative offsets (“in N minutes/hours/days”), day words (today, tonight, tomorrow, day after tomorrow, weekday names, “next Monday”), and clock times in both 12- and 24-hour form, and it stores the event with a reminder time. The browser checks for due reminders and raises a desktop notification when one is ready.

The file search only looks inside the user's own folders Desktop, Documents, Downloads, Pictures, Music, Videos, and their OneDrive versions. It searches by name, keyword, or a friendly type such as “pdf” or “image”, skips system and cache directories, and stops after a fixed number of files so a search always returns quickly. Opening a result is guarded so that only files inside those folders can be launched.

For vision, the user uploads an image and asks a question about it; both go to a local vision-language model, which answers in plain language. A related path can capture the current screen and describe it. As with text, none of this leaves the machine.

4. IMPLEMENTATION

The backend is written in Python and the front-end in ordinary web technologies. Table -1 lists the main pieces. Fig -5 shows the assistant in use, and Fig -6 to Fig -8 show the settings, appearance, and account screens.

Table -1: Main components

Layer	What we used
Backend	Flask (Python) — JSON API, session sign-in, Server-Sent Events for streaming
Language / vision	Ollama hosting a Llama-3.2-class 3B text model and a LLaVA-class 7B vision model
Storage	SQLite (users, memories, history, events, settings, faces)
Front-end	HTML, CSS, JavaScript; Web Speech API for voice; face-api.js for face login
Calendar	Custom plain-English date/time parser + reminders table with notifications
Security	PBKDF2-HMAC-SHA256 (200,000 iterations), per-user 16-byte salt, constant-time checks
Web search	SerpAPI (optional, free tier), summarised by the local model

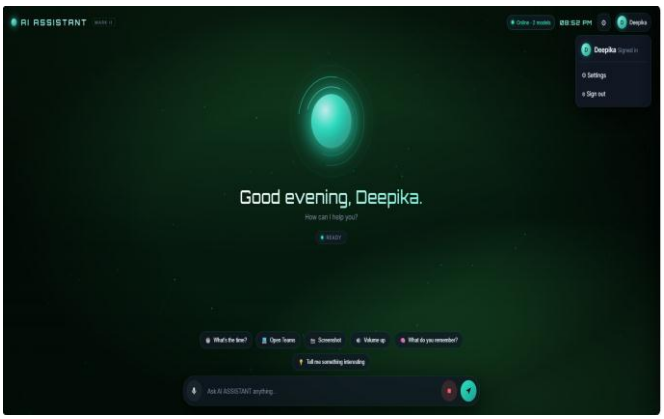

Fig -5: The main assistant screen: the personalized greeting, quick-command chips, the live model status, and the voice/text input bar.

A few implementation choices are worth spelling out. Replies from the model are streamed to the browser as they are produced, so text appears gradually instead of after a long pause, whereas a known action is answered in one go. On the first question after startup the model has to be loaded into memory, which adds a noticeable delay; after that it stays resident and responds quickly. On the first run, facts from the old system's memory file are imported once into a default profile so nothing is lost.

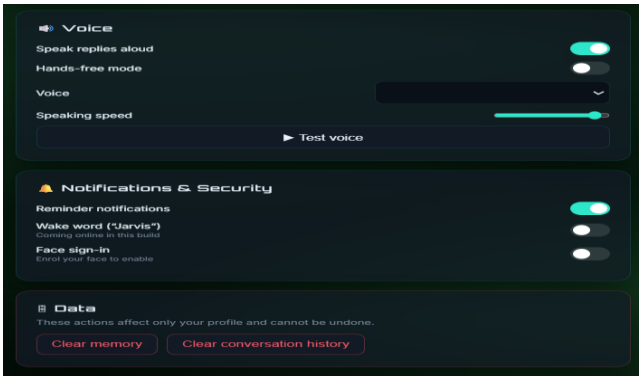

Fig -6: Settings: voice options, and the notifications and security controls including the optional wake word and face sign-in.

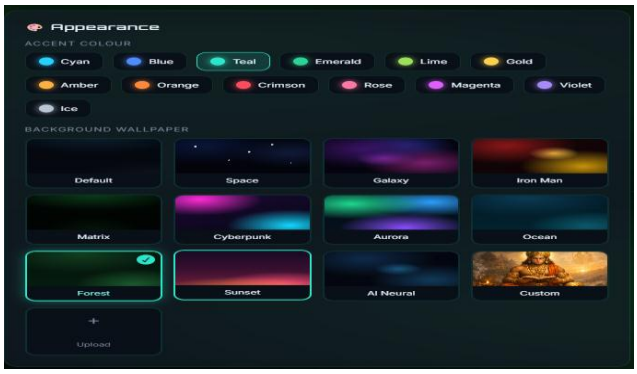

Fig -7: Personalization: accent colors and a choice of background wallpapers, saved per user.

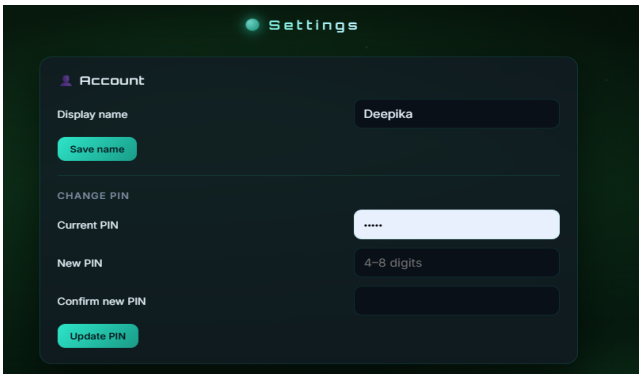

Fig -8: Account management: changing the display name and updating the PIN.

4.1 Test environment

Every number in Section 5 was measured on a single ordinary laptop with no separate graphics card: a Dell Latitude 3440 with a 12th-generation Intel Core i5-1235U (10 cores, 12 threads, 1.30 GHz base) and 15.7 GB of RAM, running Windows 11 Pro (build 26200). The backend ran on Python 3.12.10 with Ollama 0.32.1. Chat used a Llama-3.2-class 3-billion-parameter model (2.0 GB on disk) and image questions used a LLaVA-class 7-billion-parameter model (4.7 GB on disk). Because the machine has only integrated graphics, all inferences ran on the CPU. We kept it this way on purpose: it shows the assistant works on the kind of laptop a student or office already owns. The figures below are therefore close to a worst case a machine with a GPU would make the model timings much faster and leave the direct-command timings unchanged.

5. RESULTS AND DISCUSSION

5.1 How the two systems compare

Table -2 sets the original system from [1] against the rebuilt one. The differences are structural, and each one is visible in the running system.

Table -2: The original system and the rebuilt one

Feature	Original [1]	Rebuilt system
Interface	Single desktop window (Tkinter GUI)	Web app reached through the browser on localhost
Understanding	Keyword-to-command lookup	Keyword actions plus a local model for open questions
Conversation model	OpenAI cloud API	Local model via Ollama — nothing leaves the machine
Users	One, no sign-in	Multiple profiles with PIN and optional face login
Memory	None (stateless)	Per-user memory and chat history in SQLite
Reminders	Timer-based alarm	Plain-English calendar with notifications
Files	Not supported	Safe search across the user's own folders
Images	Not supported	Ask questions about an uploaded image, locally
Personalization	None	Per-user themes, wallpapers, and voice settings
Cost / privacy	Paid cloud key; data leaves machine	No paid key; all inference stays local

5.2 Speed

Table -3 reports how long each part takes on the laptop above. Each figure is an average over the number of runs shown. The volume and media rows were measured with the actual key-press to the operating system switched off, so the number reflects only the router's own work.

Table -3: Measured timing (CPU-only laptop)

Path	Operation	Runs	Time
Action	Date / time reply	30	0.01 ms
Action	Recall memory (DB read)	30	0.64 ms
Action	List calendar (DB read)	30	0.62 ms
Action	Volume / media dispatch	30	< 0.01 ms
Action	Screenshot (capture + save)	5	53.5 ms
Action	Full round trip via the API (history saved)	3	31.5 ms
File search	By type (PDF), 60 results	3	1027 ms
File search	By keyword, 12 results	3	972 ms
File search	Keyword + type, 27 results	3	852 ms
Parsing	Understand a spoken reminder	200	0.01 ms
Security	Hash a PIN (PBKDF2, 200k rounds)	10	121 ms
Security	Check a PIN	20	81 ms
Security	Match a face against 20 enrolled	200	2.6 ms
Model	First word, cold start (incl. 5.8 s load)	1	6.9 s
Model	First word, warm	5	0.91 s
Model	Full reply, warm	5	3.98 s
Model	Generation speed	5	15.8 tokens/s

Vision	Answer about a 512 x 512 image	1	45.9 s

5.3 Accuracy

Table -4 covers correctness rather than speed. For the reminder parser we wrote out 15 example phrases by hand, worked out the datetime each one should produce independently of the code, and counted a result correct only if it landed within 90 seconds of the expected time.

Table -4: Measured accuracy

What was tested	Test set	Result
Reminder date/time parsing	15 phrases	15 / 15 (100%)
Reminder title extraction	12 phrases	12 / 12 (100%)
PIN accepts the right PIN	50 attempts	50 / 50 (100%)
PIN rejects wrong PINs	50 attempts	50 / 50 (100%)
Memory recall via the model	3 facts	3 / 3 (100%)
Face sign-in success rate	—	Not run (needs camera + people)
Usability survey	—	Not run (needs participants)

5.4 Discussion

The timings say plainly why the two-part design is the right one. A known action finishes in well under a millisecond, while sending the same request through the local model costs about nine-tenths of a second before the first word and around four seconds in total. Routing something like “volume up” or “what time is it” through a model would add seconds to the commands people use most, for no gain and less reliability. Keeping the two paths apart gives instant, dependable actions and still allows real conversation where it helps.

There is an honest trade-off against the original system. The old assistant answered chat in under 1.5 seconds because it used a cloud model [1]; ours takes a few seconds on a CPU because the model runs locally. What we buy with that time is real: nothing the user types or shows is sent to anyone else, and there is no API bill. Text generation held around 15.8 tokens per second, which is fine for the short, spoken-length answers the assistant is meant to give. The vision model is the weak spot, needing about 46 seconds for one image on the CPU usable for the occasional deliberate question, but not for anything interactive, and the clearest argument for adding a GPU.

Beyond the numbers, moving from a single desktop window to a browser makes the assistant reachable from another device on the same network and easier for a non-programmer to use, and tying memory, history, and settings to a signed-in user lets it greet people by name and remember what they told it. The system is a research prototype, and we are open about its limits: face login has no liveness check and leans on the PIN behind it, the backend runs on Flask's development server on localhost, and speech recognition still uses the browser's Web Speech API, which may itself call an online service depending on the browser.

6. CONCLUSIONS

We took a keyword-driven desktop assistant and rebuilt it into a local, multi-user, browser-based system that remembers each user, answers open questions and image questions with models running on the same machine, understands plain-English reminders, and searches the user's own files. In doing so it closes the two gaps our earlier paper had left open no memory and a dependence on a cloud model and adds sign-in, personalisation, and a web interface on top. The measurements back the design: direct commands finish in under a millisecond against a few seconds for model replies, and the reminder parser, PIN check, and memory recall were correct on every test, with CPU-only vision standing out as the main thing to speed up next. Future work includes a fully offline speech path, an encrypted database and a hardened deployment, support for more languages, a phone-friendly front-end, a liveness check for face login, and a proper user study.

ACKNOWLEDGEMENT

The authors thank the Department of Artificial Intelligence and Data Science, Lingayas Institute of Management and Technology, Vijayawada, for its guidance and support during this work.

REFERENCES

[1] J. Sai Vignesh, P. Sri Lekha, Sk. Mohammad Rafi, and G. Bhanu Prakash, “Automative Desktop Assistant,” International Journal of Creative Research Thoughts (IJCRT), vol. 13, no. 10, pp. d91–d98, Oct. 2025, ISSN 2320-2882 (Paper ID: IJCRT2510370).

[2] A. Vaswani et al., “Attention Is All You Need,” in Advances in Neural Information Processing Systems (NeurIPS), 2017.

[3] H. Touvron et al., “LLaMA: Open and Efficient Foundation Language Models,” arXiv:2302.13971, 2023.

[4] A. Grattafiori et al. (Llama Team), “The Llama 3 Herd of Models,” arXiv:2407.21783, 2024.

[5] F. Schroff, D. Kalenichenko, and J. Philbin, “FaceNet: A Unified Embedding for Face Recognition and Clustering,” in Proc. IEEE CVPR, 2015, pp. 815–823.

[6] J. Lau, B. Zimmerman, and F. Schaub, “Alexa, Are You Listening? Privacy Perceptions, Concerns and Privacy-

Seeking Behaviors with Smart Speakers," Proc. ACM Human-Computer Interaction (CSCW), vol. 2, 2018.

[7] J. S. Park et al., "Generative Agents: Interactive Simulacra of Human Behavior," in Proc. ACM UIST, 2023.

[8] Ollama, "Get up and running with large language models locally," software, https://ollama.com.

[9] G. Gerganov et al., "llama.cpp: LLM inference in C/C++," software, https://github.com/ggerganov/llama.cpp.

[10] V. Mühler, "face-api.js: Face recognition in the browser," software, https://github.com/justadudewhohacks/face-api.js.

[11] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in NeurIPS, 2020.

[12] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," IETF RFC 2898, 2000.

[13] W3C / WICG, "Web Speech API Specification," https://wicg.github.io/speech-api/.