

# Vezora: Hybrid AI Personal Assistant – Project Overview

## Introduction and Concept

**Vezora** is a hybrid AI personal assistant designed to run on both Windows and web platforms for a small user group (around 5–6 test users). The assistant provides a **multimodal interface** – users can interact via text chat, voice (microphone), and even camera input for a more immersive experience. Upon activation (e.g. a wake phrase like “*Hey Vezora!*”), the assistant greets the user and is ready to assist. The vision is to create a **conversational AI companion** that can perform useful desktop and web tasks, similar to a virtual concierge or smart “Jarvis,” while respecting user privacy and budget constraints.

Vezora’s concept emphasizes a **natural, hands-free interaction**. An always-on local module listens for the wake word (if the user opts in), enabling seamless voice activation. The camera input could be leveraged for future features such as recognizing the user or scanning documents, though the core idea is that users can *talk to Vezora naturally*, and Vezora will respond – either via text or spoken replies. In summary, Vezora aims to blend the convenience of cloud AI models with on-device capabilities for a **responsive, privacy-conscious** personal assistant experience.

## Core Features (Free Tier)

The free tier of Vezora includes essential features to provide a baseline personal assistant experience without any subscription. These core features are:

- **Multimodal Input (Voice, Video, Text):** Users can issue commands or ask questions through voice (microphone) and text chat. Basic camera integration allows the assistant to receive visual input if needed (for example, scanning a QR code or recognizing a document). Voice input is processed via on-device or free API speech-to-text, and a wake word detector can run locally to continuously listen for “*Hey Vezora*” <sup>1</sup> <sup>2</sup> . This always-listening module is lightweight to minimize resource use.
- **File System Access:** Vezora can interact with the local file system for tasks like saving files or opening folders on the user’s device. Through appropriate OS-level APIs or scripting, it can create and organize files or launch file explorer windows as instructed by the user. This enables, for example, voice commands such as “Vezora, create a new folder in Documents” or “open my project file,” with the assistant executing these on the user’s Windows system.
- **Launching Applications:** The assistant can launch custom applications on the user’s machine (in the Windows environment) or open web services. For instance, Vezora can be asked to open VS Code, launch Chrome to a specific URL, or compose an email draft in Gmail. This is achieved by calling system commands or using automation scripts to start processes. (On Windows, this might use

PowerShell or shell commands; on the web version, it may utilize protocol handlers or require a lightweight local helper agent, since a web app alone cannot start native apps directly.)

- **Chat-Based Interaction with Memory:** Users can chat with Vezora in a conversational manner. Vezora maintains **contextual memory** of the conversation, meaning it remembers prior interactions (within a session and across sessions) to provide coherent, personalized responses. This memory is stored securely per user (see *Security and Privacy* below). The assistant's dialog management uses the conversation history to inform answers – for example, if a user asked yesterday “remind me to buy groceries,” Vezora can follow up on it later. The system employs efficient memory storage (potentially summarizing or chunking older dialogue when context gets long) and ensures each user's chat history is isolated and encrypted.
- **Multilingual Support:** Vezora is capable of understanding and responding in multiple languages. The underlying language model (Google's **Gemini** API in the free tier) is a state-of-the-art multipurpose model known for strong multilingual capabilities, allowing queries in languages beyond English. The assistant can detect the language of input and reply in kind, or translate when asked. Modern foundation models like Gemini are trained on diverse languages, enabling Vezora to communicate in users' preferred languages without additional setup <sup>3</sup> <sup>4</sup>. This lowers the barrier for non-English speakers and makes Vezora globally accessible from day one.
- **Encrypted Per-User Memory:** Each user's data and conversation memory are encrypted and stored on a per-user basis. In the free tier, this means that even if multiple users are testing the system, each person's notes, preferences, and history are kept in a separate, encrypted store that only that user (and the assistant when serving that user) can access. Implementation-wise, this could use an encrypted local database file for each user (if running on personal device), or encrypted entries in a central database keyed to the user. The goal is that *no plaintext sensitive data is ever stored at rest* without encryption. For example, using SQLite with SQLCipher extension can secure the local memory DB with AES-256 encryption <sup>5</sup>. That way, even if someone accessed the database file, the contents (conversation logs, to-do lists, etc.) are unreadable without the user's key.
- **Prompt-Based Gemini Integration:** At the heart of Vezora's free tier is the integration with Google's **Gemini** model via API. When a user asks a question or gives an instruction, Vezora will craft a prompt (including relevant context from memory) and send it to the Gemini API for completion. The response from Gemini is then returned to the user as Vezora's answer. Google's Gemini 2.5 models offer a free tier for developers – for instance, Gemini 2.5 Pro allows free input and output up to certain token limits, and even the expansive Gemini “Flash” model supports a limited number of free grounded prompts (e.g. ~500 requests per day with search grounding) <sup>6</sup> <sup>7</sup>. By leveraging this free allowance, the core conversational intelligence is provided at no cost. Prompt-engineering techniques will be used to ensure Gemini's output aligns with Vezora's persona and the commands (for example, telling Gemini the assistant's name and to act as a helpful personal assistant).

These core features establish Vezora as a useful assistant capable of voice and text interaction, performing local actions, and conversing intelligently *without requiring any subscription or payment*, making it accessible for initial testers.

## Advanced Features (Tiered Enhancements)

Beyond the basic capabilities, Vezora offers advanced features for power users or future premium tiers. These features are designed to add intelligence, proactivity, and cross-platform convenience, while still being **cost-effective** to implement:

- **Hybrid LLM with Fine-Tuned Models:** In addition to the cloud-based Gemini API, Vezora can utilize fine-tuned **open-source LLMs** as part of a hybrid deployment. This means we host a smaller language model (or several specialized models) that have been fine-tuned for Vezora's needs (e.g. on conversation style or specific domain knowledge). These models (such as Mistral 7B, Microsoft's Phi-2 (2.7B), or TinyLLaMA 1.1B) can run on a minimal server or even partially on the user's device. The hybrid approach allows Vezora to answer queries without always calling the external API – reducing dependency and cost. For example, **Mistral 7B** is an open 7.3B model that outperforms larger 13B models and is released under Apache 2.0 (commercial-friendly), making it ideal to fine-tune for chat <sup>8</sup> <sup>9</sup>. **Phi-2** is a compact 2.7B model from Microsoft that has shown remarkable reasoning ability for its size (matching models 10x larger) and has been open-sourced under MIT license, meaning it can be used and fine-tuned commercially <sup>10</sup> <sup>11</sup>. These models can be hosted on a server (with GPU or CPU as needed) or possibly run on user's local hardware if capable, to answer user requests directly for certain tasks. Vezora's system decides dynamically whether to use the local fine-tuned model or call Gemini based on context complexity and cost considerations (similar to routing strategies used in enterprise hybrid AI deployments <sup>12</sup> <sup>13</sup>). This provides a **cost-effective** performance boost: the open models handle routine queries or offline situations, while Gemini is still available for particularly complex or up-to-date queries.
- **Real-Time User Monitoring (Opt-in):** For users who enable it, Vezora can perform real-time context monitoring to offer proactive assistance. This might include monitoring certain aspects of user activity or environment *with full consent*. Examples include:
  - **Wake Word and Presence Detection:** The assistant can continuously listen for the wake phrase ("Hey Vezora") and also use the webcam to detect the user's presence or facial cues (all on-device for privacy). This way, Vezora can automatically activate when the user starts speaking to it or even greet the user when they sit at the computer.
  - **Activity Monitoring:** Vezora could optionally watch for specific application usage or system events. For instance, if the user opens their email every morning, Vezora might proactively say "I see you're checking email; do you want me to summarize new messages?" (This would rely on OS-level event hooks or a small background process reading the active window titles, etc. – again, only if the user permits such access).
- **Health and Break Reminders:** Using the webcam or system idle time, Vezora could notice if the user has been working continuously and suggest a break, or detect if the user looks tired and recommend a rest. This blurs into wellness monitoring. In implementing real-time monitoring, **privacy and user control are paramount**. All such features are off by default and clearly explained, and any processing of camera/mic outside explicit queries is done locally. This aligns with a hybrid approach where on-device NLU handles wake word and simple commands instantly <sup>1</sup> <sup>2</sup>, without sending constant data to cloud. The *opt-in monitoring* can make Vezora more of a proactive digital assistant (similar to how smart speakers offer always-listening wake words), but with the user's comfort levels respected.

- **Scheduled Automation Tasks:** Vezora can act as an agent that not only responds to immediate commands but also performs **scheduled tasks** automatically. Users can instruct Vezora to set up routine automations like “Check my email every day at 8 AM and give me a summary” or “Every Friday, backup my project folder to the cloud.” Once set, these tasks run at the scheduled times (even if the user doesn’t explicitly prompt them). For instance, an email-check job might use Vezora’s integration with Gmail’s API to fetch new messages and then use the LLM to summarize them, delivering the summary via a notification or morning briefing. Similarly, Vezora could scrape a particular website daily for updates, log time entries, or perform system maintenance tasks. The scheduling could be achieved with a lightweight scheduler (like cron jobs or a scheduling library in the backend). Because only 5–6 users are involved, the load is small; tasks can be queued or run in separate threads/services without significant infrastructure. **Opt-in and transparency** are again key – the user must authorize any access (e.g., storing email API tokens securely if needed) and should be able to view or cancel scheduled automations easily. This feature greatly enhances personal productivity, essentially letting Vezora handle repetitive chores in the background.
- **Voice Synthesis (Speech Output):** In the advanced tier, Vezora not only listens but also *talks*. Using text-to-speech technology, it can generate spoken responses, giving the interaction a more natural, human-like feel. This is helpful for hands-free use (e.g., Vezora reading out an email while the user is away from the screen). The voice synthesis can be implemented using **open-source TTS models** to avoid licensing costs – for example, projects like Mozilla’s TTS or Coqui TTS provide high-quality neural speech synthesis that can run locally. Another emerging option is the “Bark” model for realistic voice generation. For efficiency, the voice can be generated on the user’s device (if it’s capable) or on the server side and streamed to the client. There are also lightweight options like using the OS’s built-in voices or free tiers of cloud TTS (Google Cloud Text-to-Speech offers some free quota monthly). Given the ₹0 budget preference, offline TTS is ideal: e.g., the **Mimic 3** engine (from Mycroft AI) can run fully offline with pre-downloaded voice models and is optimized for fast, privacy-friendly synthesis <sup>14</sup>. By integrating such a solution, Vezora can speak with a pleasant voice without incurring costs per character. Users can enable or disable voice output as they prefer.
- **Cross-Platform Memory Syncing:** A key advanced feature is that a user’s personal assistant experience is seamless across devices. Whether a user is interacting with Vezora on their Windows PC or through the web interface (say, on a laptop or tablet), their assistant “remembers” their context and data. This is accomplished by **cloud-syncing the encrypted user memory**. The memory database or knowledge store (notes, preferences, conversation history, to-do lists, etc.) is stored in a central repository (cloud database or secure storage) so that when a user logs in from any device, Vezora fetches their data (decrypting it locally using the user’s key). For example, if a user set a reminder or had a conversation on their work PC, and later uses the web app on their home laptop, Vezora can pick up right where they left off – “As we discussed earlier, I’ve prepared that to-do list for you...”. The sync is **secure**: only encrypted data travels over the network, and decryption happens client-side. Implementing this could involve using a small cloud database (see *Architecture* below) accessible via an API. We will use unique user accounts or keys so each user’s data is isolated. This feature greatly improves convenience – users get a consistent assistant experience anywhere – while still ensuring no one except the user (and their instance of Vezora) can read their memories.

All these advanced features are designed to add value for users in a **cost-effective way**. By leveraging local processing and open-source components, we minimize ongoing costs while providing capabilities often

found only in expensive AI assistants. Users can opt into the features they need, keeping the experience personalized and privacy-respecting.

## Architecture and Workflow

Vezora's architecture follows a **hybrid cloud-local model** that balances performance, cost, and privacy. The system is composed of modular components and services, each handling a specific aspect of the assistant's functionality. Below is an outline of the architecture and how data flows through the system:

**1. Frontend Interface (Windows and Web):** The user interacts with Vezora via either a desktop application (for Windows) or a web app. To maximize code reuse and ease of development, the frontend could be implemented as a web-based interface (HTML/CSS/JS) that runs in a browser or an Electron container on Windows. Tools like **Lovable.dev** (an AI-powered no-code builder) can accelerate creating this UI, generating a chat window, controls for voice/camera, and even basic backend hookup with little coding <sup>3</sup>  
<sup>15</sup>. The interface includes: - A chat area (showing conversation history, answers, etc.). - A microphone toggle (for voice input) and speaker toggle (for voice output). - Possibly a small video preview if camera input features are used. - Buttons for common commands or to open certain apps. - A user login/auth system (to identify user and load their encrypted memory from the DB).

**2. On-Device Services (Local Layer):** Running on the user's machine (especially in the Windows app scenario) are several lightweight services: - **Wake Word Engine:** Always listening for "Hey Vezora" using a tiny model. This could be a very small footprint model (few MB) that runs continuously with minimal CPU. When it detects the phrase, it signals the system to start listening for a command <sup>1</sup> <sup>2</sup>. - **Speech-to-Text (STT):** Converts spoken input to text. For quick offline processing, an embedded STT like Vosk (open source) can handle this. Alternatively, the audio can be sent to a cloud STT if online, but local is preferable to avoid cloud costs for every command <sup>16</sup>. The STT output text is then forwarded to the language understanding module. - **Local NLU/Micro-LLM:** A **Natural Language Understanding** module or a small "micro-LLM" resides here to handle basic commands locally. This could be a distilled version of an LLM or even a rules-based intent recognizer. Its job is to parse the user's request and decide if it's something simple that can be answered/executed locally, or if it needs the heavy cloud LLM. For example, commands like "Open Gmail" or "What's the time?" don't need an AI model call – they can be caught by intent rules or a small model. The on-device NLU can also provide immediate feedback (like "Sure, on it!") for simple tasks, improving responsiveness. Critically, it **routes queries:** if the user's question is complex ("Explain quantum computing..."), the NLU will pass it to the cloud/backend LLM; if it's trivial ("Volume up" or "Thanks Vezora"), it may handle it directly <sup>17</sup>. This intelligent triage ensures that we **only use cloud resources when necessary**, cutting down latency and cost <sup>18</sup> <sup>19</sup>.

- **Device Controller:** This service receives high-level commands (like "open Chrome and go to Gmail") and interfaces with the OS to execute them. On Windows, this might be implemented in a Python or Node.js backend that the UI calls (for example, if using Electron, a Node process can run shell commands). It has access to launch applications, open files, etc., restricted by what we allow for safety. For web-based usage (where such access is not possible directly), we might require a minimal local agent or limit these features to the desktop app for now.
- **Local Memory Cache:** The assistant maintains a local cache of recent conversation context and user-specific data (decrypted in memory). When online, it can sync with the central DB, but at runtime the conversation history is kept ready locally to include in prompts. This cache might include vector

embeddings for recent topics for quick similarity search if needed (for example, to recall something said earlier without always sending the entire history to the LLM).

**3. Cloud/Server Backend (Cloud Layer):** The cloud components handle the heavy lifting and data synchronization: - **Primary LLM API:** For complex queries or creative answers, the user's request (plus context) is forwarded to the cloud. In the free tier, this is the **Google Gemini** API. Our backend will call Gemini with a well-structured prompt. The prompt likely includes a system message defining Vezora's role and personality, the conversation context (pulled from memory, within token limits), and the user's latest query. Gemini processes this and returns a completion (answer), which our backend relays back to the client interface. - The usage of Gemini is optimized by the earlier local NLU step: only queries deemed complex go here. This saves money and time – e.g., the local model might catch 30% of interactions, meaning 30% fewer API calls <sup>17</sup>. This hybrid strategy reflects approaches in industry where on-premise models handle easy cases and cloud is used for hard cases <sup>20</sup> <sup>13</sup>. - The free tier allowances of Gemini (e.g. free input/output tokens to a certain extent <sup>21</sup> <sup>22</sup>) mean we expect no cost for moderate usage. Should usage approach limits (unlikely with 5–6 testers), the system could gracefully fall back to the open-source model or a smaller “Gemini Lite” without charge.

- **Open-Source LLM Service:** In advanced tier mode, our backend may also host an instance of a fine-tuned open model (like Mistral 7B instruct or Phi-2). This could be a **REST API** running on a cloud VM or container. If using Hugging Face Spaces for deployment, for example, we might run the model with a framework like `ctransformers` to serve it on CPU <sup>23</sup>. The assistant can decide to query this service instead of Gemini for certain requests (perhaps based on a cost/latency policy or user preference like “offline mode”). The model might be slightly slower or less powerful than Gemini, but it ensures continuity even if external API is down or if we want to avoid sending data out. It also can be fine-tuned to Vezora's style to give more customized responses.
- **Web App Backend:** If the user is on the web interface, a backend server is needed to intermediate between the web client and the local/OS functions. This backend (which could be the same server handling LLM API calls) would expose endpoints for things like “/sendPrompt” (to send prompt to LLM), “/getMemory” (to retrieve user memory from DB), etc. The web client then interacts with this backend via HTTPS. We must secure these endpoints with authentication tokens since user data is involved.
- **Modular Services:** The architecture is broken into **modular services** so each part is maintainable and can be scaled or replaced independently. For example, the speech recognition, text-to-speech, and vision (camera) processing can be separate modules. A **service bus** or simple message passing could connect them. In a minimal implementation, modules can run as threads or processes on the same machine; in a more scalable design, they could be containerized microservices. This modularity ensures that enabling/disabling a component (say, turning off voice output) is straightforward, and one module's updates won't break others. It also allows mixing and matching – e.g., if later switching from Google STT to another service, or adding a new module for OCR on images, it can slot in without redesigning the whole system.

- **Memory and Session Database:** A central database stores persistent data such as encrypted user memories, profiles, and session info. We seek a **lightweight yet secure DB** solution:

- For simplicity and security, an embedded database like **SQLite** with encryption can be used. SQLite is file-based, requires no separate server process, and can handle multiple users' data in separate tables. Using the open-source **SQLCipher** extension, we can encrypt the entire database or specific fields, so that the data at rest is protected <sup>5</sup>. Each user's records could be encrypted with a key derived from their login credentials (meaning even if someone got access to the DB file, they couldn't read individual data without each user's key).
- Alternatively, since cross-device sync is needed, a cloud-hosted database might be used. A small **PostgreSQL** instance (e.g., on Supabase or Neon) could serve as the central store. Postgres offers robust security features (and can be configured with Transparent Data Encryption on some platforms). We would still encrypt sensitive fields at the application level. Given the low user count and data volume, the free tier of such DB services is likely sufficient (many offer ~10MB to 100MB storage free, which is plenty for text history). A NoSQL option like **Firestore** or **Deta Base** could also work for simplicity, but an SQL database is convenient for structured querying of sessions and logs.
- The database stores things like user account info (username, salted hash of password for auth), preferences toggles (like whether they enabled real-time monitoring), and the conversation logs or vector embeddings. The **principle of least knowledge** is applied: only the necessary info is stored (for example, we might not log raw audio or images at all; only derived text). And wherever possible, data is encrypted per user such that the server side alone cannot decrypt it (the decryption key is provided by the user upon login). This ensures *only the user can decrypt their memory*, fulfilling the privacy design.

**Workflow Example:** To illustrate how these pieces work together, consider a user saying: *"Hey Vezora, open Visual Studio Code and tell me if I have any new emails."* 1. The wake word is caught by the local module (no cloud involved). The STT locally transcribes the full command to text: "open Visual Studio Code and tell me if I have any new emails." 2. The local NLU parses this. It identifies an *actionable command* ("open VS Code") and a *question* ("do I have new emails?"). The NLU might know how to handle the app launch itself, so it triggers the Device Controller to open VS Code immediately (providing immediate feedback to the user: "Launching VS Code..." handled locally). 3. For the email query, it recognizes this likely needs to check an external service (Gmail API) and perhaps use the LLM to summarize. The NLU module decides this requires cloud and possibly the LLM's help. So it formulates a structured task for the cloud layer. 4. The backend receives a request: it may first call a Gmail integration (if user has linked their account) to fetch, say, the count of new messages or the subjects. The backend then crafts a prompt for the LLM: e.g., *"The user has 5 new emails, subjects are A, B, C... Summarize if any seem important."* and sends this to either the open model or Gemini, depending on configured logic. Suppose it uses Gemini for the best quality summary. 5. Gemini returns a summary answer about the emails. The backend sends this response back to the client app. 6. Vezora then speaks the answer aloud via the TTS module: "You have 5 new emails. None look urgent – a newsletter, a couple of project updates, and a notification from GitHub." It also displays this text in the chat window for reference.

Throughout this, the session context (what the user asked, what was done) is logged in the memory (in encrypted form) so that later Vezora can recall that VS Code was opened or that it already notified the user about their emails.

This **multi-API integration** (combining local OS APIs, web service APIs like Gmail, and LLM calls) is orchestrated by Vezora's modular backend. Each skill or integration is developed as a module (e.g., an EmailChecker module, a FileLauncher module) which the main logic calls upon. The architecture being hybrid means if the internet is down or the cloud is not reachable, Vezora won't be entirely useless – it can still perform local commands and simple Q&A using the on-device model for some answers <sup>24</sup> <sup>25</sup> . Conversely, when online, it leverages the full power of cloud AI. This design offers **flexibility, efficiency, and resilience**, all while keeping user data safe and local whenever possible.

## Security and Privacy

Security and user privacy are fundamental in Vezora's design. The assistant deals with potentially sensitive personal information, so we implement multiple layers of protection and give users control over their data. Key aspects include:

- **Encrypted User Memory:** All user-specific data (conversation logs, personal notes, API keys, etc.) is encrypted at rest. Each user's memory is encrypted with a unique key. If the data is stored on the user's device (Windows app scenario), it can be encrypted on an OS-provided secure storage or an encrypted local database file. If stored on the cloud DB for syncing, the encryption is done **before** sending to the cloud (client-side encryption) so the server never sees plaintext. For example, using SQLite + SQLCipher means the entire DB file can be encrypted with a passphrase only the user knows <sup>5</sup> . Another approach is field-level encryption: e.g., each message could be individually encrypted with a symmetric key derived from the user's password. The bottom line is even if an attacker got hold of the database or if the cloud host was compromised, the content remains gibberish without the key. Only the authenticated user (or their device) can decrypt their memory. This ensures that *each user truly owns their data*.
- **Local-First Processing:** Wherever feasible, Vezora processes data locally on the user's device rather than sending it to external services. This is both for privacy and cost-saving. Sensitive inputs like the user's voice and camera feed are handled on-device. For instance, wake word detection and speech transcription can run offline, meaning your microphone's raw audio isn't continuously streamed to the cloud <sup>16</sup> <sup>26</sup> . If the user uses the camera to show a document, we can run local OCR or analysis so that images aren't uploaded elsewhere. By **keeping sensitive operations local**, we reduce exposure of personal data. The cloud is used only when necessary for complex queries or external info, aligning with the hybrid philosophy *"process sensitive info locally, offload generic tasks to cloud"* <sup>27</sup> .
- **Opt-In Features and Transparency:** Features like real-time monitoring via camera/mic, or accessing third-party accounts (email, calendar), are strictly opt-in. Users must explicitly grant permission and can revoke it at any time. Vezora will provide clear notifications – e.g., an indicator when the microphone is actively listening for the wake word, or logs of what it monitored/scheduled. This approach ensures users are never caught by surprise by background activities. If a user opts in to continuous listening, it's made clear that audio is being analyzed locally for the wake word. If they opt in to face recognition for greeting, the facial data is processed on-device (possibly using a library that does face encoding locally) <sup>28</sup> and not sent out. We follow the principle that **the user is in control** of their data and device.

- **Secure APIs and Credential Handling:** When integrating with external services (like Gmail, calendar, etc.), we use OAuth or secure token mechanisms. Tokens are stored encrypted in the user's memory store. They are only used when the user triggers those functions and are never exposed in plaintext in logs. Communication with cloud APIs (Google Gemini, etc.) is over HTTPS with proper authentication (API keys or tokens stored securely). We will implement **rate limiting and input validation** on any exposed endpoints of the backend to prevent abuse or injection attacks. For example, if we have a `/executeCommand` API for launching apps (for web clients), it will have an allowlist of safe commands to avoid arbitrary execution, and it will be authenticated per user to prevent one user triggering actions on another's device.
- **Database Security:** The chosen database solution will be secured with best practices. If using a SQL database server, it will run in a protected environment with strong credentials, accessible only by the application. If using an embedded DB like SQLite on the server, file permissions are locked down. We implement **backups** of the database (encrypted backups) to prevent data loss, but even backups remain encrypted. Additionally, we might employ a lightweight hashing or indexing on encrypted content to allow some retrieval without decrypting everything (for example, storing a hash of conversation IDs to quickly fetch the right blob, without revealing content). For user authentication, we store salted password hashes (using algorithms like bcrypt or argon2) – not the actual passwords.
- **No Oversharing of Data with LLM:** Another privacy measure is careful prompt engineering to avoid sending sensitive data to the LLM unless necessary. For instance, if a user's query is "What's the balance of my bank account?" and if Vezora has that info stored (say via a linked account), we shouldn't just naively send the raw number to Gemini for an answer. Instead, the local logic could handle such a question directly, or only send minimal context needed. We avoid putting things like personal names, addresses, or large text from user documents into a prompt to a cloud LLM unless the feature explicitly demands it (and user knows, like asking "summarize this private document" – in which case perhaps the open local model could be used as an alternative). Keeping more tasks on-device when privacy is a concern is a strategy supported by hybrid models <sup>27</sup>.
- **User Data Ownership and Export:** Users will be given the ability to export or delete their data. Since only they can decrypt their memory, if they choose to delete it, we ensure all encrypted records are removed from the DB. If they want an export, Vezora can decrypt the memory with their key and provide it in a human-readable form. This is good for transparency and for compliance with data protection principles (even if this isn't a commercial product yet, it's good practice to handle data as if under GDPR/CCPA – e.g., honoring deletion requests, etc.).
- **Session Security:** If using the web app, sessions will be managed securely (likely JWT tokens or secure cookies). We will implement short token lifetimes and idle timeouts so if a user leaves the interface open, it auto-locks after a while, requiring re-authentication to continue, to prevent someone else using their assistant. The Windows app can similarly have a lock mechanism (optionally require a PIN or Windows Hello auth to access Vezora after idle).

By combining strong encryption, local-first design, and user-centric permissions, Vezora aims to be **secure by design**. The architecture inherently limits data exposure – e.g., always-listening modules run offline, not on cloud servers <sup>16</sup>. And any cloud communication is encrypted in transit (HTTPS) and minimal. In effect,

each user has their “own little vault” of AI knowledge that only they and their AI assistant can open. This focus on privacy will be a key differentiator, reinforcing user trust in Vezora.

## Technology Stack Suggestions

To implement Vezora efficiently and within budget, we propose a technology stack that leverages both state-of-the-art AI services and open-source tools. Here are the key components and their justifications:

- **LLM Backbone (Free Tier): Google Gemini** via Google’s AI API will serve as the primary language model for the free tier. Gemini (notably versions like *Gemini 2.5 Pro or Flash*) is Google’s latest-generation model known for excellence in coding, reasoning, and multimodal understanding. It’s accessible through the Google AI Studio/API with free usage quotas <sup>21</sup> <sup>22</sup>, which makes it ideal for a low-budget project. By using Google’s API, we get a high-performance LLM (comparable to GPT-4 in capability) essentially for free up to certain limits. This will handle natural language understanding, conversation, and reasoning tasks when the local model isn’t sufficient. The integration will likely be via REST calls with an API key that Google provides (developers can obtain a free key and maybe a quota of a few thousand tokens per day at no cost).
- **Fine-Tuned Models (Hybrid AI):** For the advanced tier or as a supplement, we will use **open-source models** like *Mistral 7B*, *Phi-2 2.7B*, or *TinyLLaMA 1.1B*. These can be fine-tuned on our custom data (if needed) and hosted without expensive infrastructure. Specifically:
  - *Mistral 7B*: chosen for its strong performance at relatively small size. It’s easy to fine-tune (supports techniques like LoRA) <sup>9</sup> and runs faster than larger models. We can host it on a modest cloud VM or even use CPU inference with quantization. Mistral’s Apache 2.0 license and superior benchmark results <sup>29</sup> make it a top pick.
  - *Phi-2 (2.7B)*: extremely lightweight, could even run on some user machines. Since Microsoft open-sourced it under MIT, it can be commercially used <sup>11</sup>. We might fine-tune Phi-2 to better handle dialogue (since it’s somewhat QA-oriented out-of-the-box). Its small size means it can be served on CPU easily – perhaps even directly within the app for offline mode.
  - *TinyLLaMA (1.1B)*: this model is tiny, designed to be trained on huge token counts <sup>30</sup>. It’s not very powerful alone, but fine-tuning it for specialized tasks (like quick command recognition or offline question answering) could be useful. It could run in-memory on virtually any device, providing a fallback when no internet is present.

Fine-tuning these models will be done off-device (see next section), and the resulting model binaries (possibly quantized to 4-bit or 8-bit for efficiency) will be deployed. We can host them on **Hugging Face Spaces** (which offers free CPU hosting with decent RAM) <sup>23</sup> or on a minimal cloud service like an AWS t2.medium instance. Another approach is to use **Google Colab** or **Kaggle Kernels** to serve the model for free, though those aren’t 24/7 reliable. HuggingFace Spaces with a CPU container is attractive: it’s free and gives 16GB RAM and 2 vCPU – enough for a quantized 7B model, albeit with slower response (perhaps ~30+ seconds a response) <sup>31</sup>. For initial testing, that speed is tolerable. If needed, we could attach an inexpensive GPU on a pay-as-you-go basis for heavy use.

- **Frontend Development:** The combination of **Lovable.dev** and **Cursor** is suggested for building the frontend and overall application quickly. **Lovable** is a no-code/low-code AI builder that can generate full-stack app components from conversational prompts and designs <sup>3</sup> <sup>32</sup>. We can use it to

scaffold the UI – for example, design a chat interface with a sidebar for settings, and Lovable will produce the responsive HTML/CSS and even setup basic backend routes. It also can handle deployment (Lovable Cloud) and databases for simpler use cases, which might cover some of our needs (it auto-provisions a PostgreSQL database and auth if needed) <sup>33</sup>. This reduces manual coding and ensures a professional-looking app with minimal effort. **Cursor**, on the other hand, is an AI-assisted IDE (built on VS Code) that helps write and refactor code faster <sup>34</sup>. The development team can leverage Cursor for implementing custom logic (like the Python scripts for device control or the integration code for APIs). It will speed up writing boilerplate, tests, and even the tricky parts of multi-API orchestration by providing smart code completions and suggestions. Essentially, Lovable covers the *no-code frontend* while Cursor accelerates the *pro-code backend*.

- **Programming Languages & Frameworks:**

- For the **Windows client app**, a cross-platform framework like **Electron** or **Tauri** can be used to wrap the web frontend into a desktop application. Electron allows Node.js integration (which we can use to call OS commands for launching apps). Tauri is more lightweight and secure, though slightly less mature for heavy integration – it could also be considered, especially if using Rust for system calls. Given our need for quick development, Electron + React (or even a simple HTML/JS stack) might be the fastest route, possibly generated via Lovable's export.
- For the **backend server**, we have flexibility. If Lovable is used, it might generate a Node.js/Express or Python/FastAPI backend automatically. We can stick with whatever integrates well. A Node.js backend could be handy for using the same language across frontend and backend (JS/TypeScript). Node also has libraries for system operations (though Python is often more straightforward for OS tasks on Windows). Another approach is to use **Python** for backend (especially to interact with local hardware using libraries like `subprocess` for launching apps, or `pywin32` for Windows automation). Python also has excellent support for ML integration if we wanted to embed some local model. Since this is a prototype with few users, performance differences are minor, so we can choose the language the team is most comfortable and productive in.
- If using Lovable's one-click deploy, it might favor Node/Next.js or similar stack. We could use that to deploy the web app easily on their cloud (free for initial usage).

- **APIs and Integrations:** We plan to integrate various third-party APIs for added functionality in advanced features. This includes:

- Email (Gmail API or IMAP for other providers) for reading/sending emails.
- Calendar APIs (Google Calendar, Outlook) for scheduling or reminders.
- Possibly task management APIs (like Todoist or Notion) if we integrate to-dos.
- Web scraping or RSS feed reading for certain scheduled info (news summary etc.).
- We will use appropriate SDKs for these in our backend. Many have free tiers for low usage which suits 5–6 users.

- **Voice and Vision Libraries:**

- For **speech recognition**: likely **Vosk** (which has offline models for many languages) or **Whisper** (OpenAI's model, but that might be heavy to run fully offline; however, small variants or Whisper API free tier if any could be used). Windows also has a built-in speech API that could be tapped for free.
- For **text-to-speech**: as mentioned, **Coqui TTS** or **Mimic3** for offline voices. If willing to use an online service, Google's TTS or Microsoft Azure's TTS have free quotas (e.g., Azure gives some million characters free per month). But offline ensures no ongoing cost. We could also allow plugging into OS-native TTS (Windows has SAPI). As a quick solution, even using Python's `pyttsx3` (which uses SAPI on Windows) could produce a basic voice without extra dependencies.
- For **camera input processing**: if we add say OCR, we can use **Tesseract** (open source OCR engine) for reading text from images, or simple CV tasks with OpenCV. If doing face detection, OpenCV or a lightweight model (like YuNet or even Dlib) could run locally to just detect a face or recognize a specific user's face (with their training).
- **Database**: As discussed, **SQLite** with encryption is a top choice for simplicity. If a server-based DB is needed for multiuser, then **PostgreSQL** (possibly via Supabase for ease of setup) is recommended. Postgres is reliable and lightweight enough for our scale. Supabase's free tier includes a hosted Postgres with 500MB and 2 CPU, which is plenty. It also provides auth and storage if needed, though we likely handle auth separately. Another lightweight DB could be **DuckDB** (embedded analytics DB) but that's overkill; or **MongoDB Atlas** free tier if we preferred NoSQL (but relational fits our needs with user tables, etc.).
- If Lovable is used for backend, it might set up a Postgres automatically. That could actually save us time – we'd then just ensure to enable encryption or encrypt data before saving.
- **Deployment and Hosting**: For hosting the web component and backend, we prefer **free or very low-cost services**:
  - **Hugging Face Spaces**: as mentioned, for hosting the ML model API (and potentially could host a simple Gradio interface or FastAPI for the assistant, though it's mainly for demos).
  - **Vercel / Netlify**: could host the web front-end (static files or Next.js app) on their free plan. Vercel, for instance, could even host a serverless function for the backend API calls (like using Next.js API routes or using their Serverless Functions) which might cover our needs within free limits.
  - **Railway.app** or **Fly.io**: these offer free tiers for running small containers or servers (Railway's free tier is limited by \$5 credit, Fly has a certain amount of free VM hours). We could deploy our Node/Python backend here if needed, staying within the free usage by shutting down when idle, etc.
  - **Deta Space**: an option for free hosting of Node or Python microservices with persistent storage; could be viable for the backend and DB with small scale.
  - **Oracle Cloud Free Tier**: provides always-free small VMs and even GPU time. Oracle's free Ampere ARM VM can run a lightweight server 24/7 at no cost, which might be perfect for our central server and even running the small LLM (if compiled for ARM).
  - Given cost constraints, we will attempt to use these free services first and only consider paid options if usage grows. The system is designed such that it can run *hybrid* – some parts on user's machine – to reduce how much we must host centrally.

In summary, the tech stack centers on using **Gemini for strong AI capabilities (free), fine-tuned open models for customization (hosted cheaply)**, and modern app development tools (**Lovable, Cursor**) to rapidly build a cross-platform solution. We favor open-source and free-tier-friendly technologies at every

layer: from database (SQLite/Postgres) to deployment (HF Spaces, Vercel) to libraries (Vosk, Coqui, etc.). This stack is not only budget-conscious but also allows flexibility to pivot (e.g., swap out Gemini for another API if needed, or scale up the open-source model hosting if more users join). It ensures that for our 5–6 users, the experience will be smooth without incurring significant costs.

## LLM Fine-Tuning Strategy

To enhance Vezora's intelligence without relying solely on third-party APIs, we plan to fine-tune open-source language models on specific data or behaviors. The fine-tuning strategy is geared towards using **low-cost resources (like free Colab GPUs)** and efficient techniques to stay within budget while achieving good model performance.

Here's the approach:

- **Selecting the Base Models:** As discussed, the models of interest are Mistral 7B, Phi-2 (2.7B), and TinyLLaMA 1.1B. Each serves a purpose:
- *Mistral 7B:* likely our main target for fine-tuning into a chat/instruct model. It's already powerful and even has an available chat-tuned variant (Mistral 7B Instruct) <sup>35</sup>. If that suits our needs, we might use it as-is; otherwise, we fine-tune on our own prompts to better align it with Vezora's persona or to add knowledge of specific user commands.
- *Phi-2 2.7B:* we'd fine-tune this to improve its chat capabilities and possibly to compress some of Vezora's knowledge into it (like frequent Q&A pairs, or understanding how to execute commands rather than just answer questions). Being small, Phi-2 is **cheap to fine-tune** and even just 5-10 epochs on a small dataset could teach it Vezora-specific behavior. With its new open license, we can deploy it freely <sup>11</sup>.
- *TinyLLaMA 1.1B:* this model might be fine-tuned on extremely small hardware. The project is designed to pretrain on lots of data; we could adapt it to conversational fine-tuning. This could serve as the on-device mini model for quick responses or classification tasks.
- **Data for Fine-Tuning:** We will gather or generate a **small fine-tuning dataset** tailored to Vezora. This might include:
  - A set of typical user commands and desired responses (for example: *User: "Open the browser" -> Assistant: "Sure, opening Chrome now."*).
  - Conversations that show the style/personality we want (polite, slightly witty, always helpful).
  - Multilingual Q&A pairs to bolster non-English performance if needed.
  - Some knowledge about the user's environment (for example, if we want the model to know what VS Code is or how to respond to certain OS queries without going to the internet).

We can also leverage existing open instruction datasets as a base (there are many, like Dolly, OIG, etc.) and just add a Vezora-specific layer on top.

- **Fine-Tuning Method:** We will use **parameter-efficient fine-tuning** techniques such as **LoRA (Low-Rank Adaptation)** and 4-bit quantization (QLoRA) to drastically reduce the resource requirements <sup>36</sup> <sup>37</sup>. This way, we don't need to update all 7B parameters of Mistral; we only train small adapter matrices (millions of params) which can be done on a single GPU.

- **Google Colab** (free tier) provides access to GPUs like Tesla T4 for limited durations. It's been demonstrated that with LoRA and 4-bit quantization, one can fine-tune models like LLaMA-7B on a Colab free instance <sup>38</sup> <sup>39</sup>. We can follow similar guides – for instance, using the **PEFT** library and **bitsandbytes** for 4-bit training. This allows a 7B model to fit in about ~4GB of GPU memory during training, which is feasible on a T4 (16GB).
- We can split our fine-tuning tasks: Fine-tune Mistral on one Colab session (taking maybe a couple of hours for a few epochs on a small dataset), fine-tune Phi-2 on another (Phi-2 with 2.7B params will be even faster, possibly under an hour with LoRA), etc. If Colab limits runtime (free sessions often cap at 1-2 hours), we can save checkpoints and resume if needed.
- Additionally, there are community resources: sometimes one can use **Kaggle notebooks** or **Google Research TPU free tiers**, but likely Colab suffices. There's also **Hugging Face free training** on small models (they sometimes provide free Cloud credits for model training in events, but we might not rely on that).
- **Testing and Iteration:** After fine-tuning, we will evaluate the models on some test prompts to ensure they behave as expected (e.g., they follow instructions to launch apps only when appropriate, they respond in the desired tone, etc.). If issues arise (like the model hallucinating or giving unsafe answers), we might further refine with additional data or apply some guardrails (could fine-tune a bit more on a safety dataset or use the NLU layer to filter the model's outputs).
- **Deployment of Fine-Tuned Models:** Once ready, the fine-tuned model weights (or LoRA adapters) will be uploaded to a repository (maybe a private Hugging Face repo or stored on our server). We will then load them in our inference server. For example:
  - Use **ctransformers** or HuggingFace's `AutoModelForCausalLM` with 4-bit quantization to load Mistral-7B with our LoRA merge on CPU <sup>40</sup> <sup>41</sup>. This library is very memory efficient and written in C++ so it's fast on CPU. According to one demo, a 4-bit quantized 7B running on CPU can generate ~150 tokens in about 60 seconds on 2 cores <sup>31</sup>. On a better CPU or with multi-threading (ctransformers allows setting number of threads), we might improve that. And if performance is a concern, we could later move to a low-cost GPU instance.
  - For Phi-2 or TinyLLaMA, their small size means even unquantized they might run quickly on CPU. We will likely quantize them too for consistency and to allow running possibly on user side if needed (e.g., a 2.7B 4-bit model can run on a high-end smartphone theoretically <sup>42</sup>, which speaks to how accessible these are).
- **Serving Multiple Models:** Our system might host two models simultaneously: one smaller (fast) and one larger (smarter). For example, host Phi-2 as a quick response model and Mistral for more complex tasks. We can route queries accordingly. This is similar to having a local "tier-1" model and a cloud "tier-2" model in enterprise hybrid solutions <sup>13</sup>. Since all our open models are relatively small, a single VM could probably load them sequentially as needed (not run concurrently to save memory).
- **Monitoring Costs of Fine-Tuning:** Using free resources like Colab means our training cost is ₹0. It requires some manual effort (or writing a script to utilize Colab's API if any). If needed, we could consider a very short-term rental of a GPU (services like **RunPod** or **Paperspace** allow renting A100 or 3090 GPUs by the hour). For instance, a RunPod A10 (24GB) might cost around \$0.4 per hour; if we

needed 2 hours, that's under \$1 – negligible in our ₹10k budget. But likely, we can avoid this by smart use of free tools. The model sizes are small enough to not require lengthy training.

In summary, our fine-tuning strategy is **small-scale, targeted, and frugal**. We focus on *incrementally improving open models* to align with Vezora's tasks. By leveraging LoRA and free GPU instances, we avoid major expenses. The outcome will be a set of custom models that we own (no recurring API costs) which can handle a good chunk of interactions – ensuring that even if the external Gemini API becomes costly or unavailable, Vezora remains functional and intelligent on its own. This strategy also allows rapid experimentation: we can fine-tune, test, and iterate quickly in the constrained environment, and any significant gains (like a model that can handle 80% of requests locally) directly translate to cost savings and better privacy for users.

## Hosting Strategy

With both local and cloud components, our hosting strategy must ensure reliability for the test users while keeping costs near zero. We propose a **hybrid deployment** approach, where some services run on users' devices and others on the cloud, choosing **economical/free cloud options** for the latter.

Key points of the hosting strategy:

- **Local vs Cloud Split:** We delineate which parts run locally (on each user's machine) and which run in the cloud:
- Running locally on Windows: the wake word listener, STT, TTS, device control, and possibly even the small LLM model (Phi-2 or TinyLLaMA) if we find it feasible to package it. The Windows app installer could bundle a quantized 2.7B model and necessary runtime (which might be a few hundred MB – possibly acceptable). If not bundling a model, the local app will at least handle the multimedia I/O and send text queries to the cloud.
- Running on cloud: the main LLM service (Gemini API usage and/or Mistral model inference), the central database, and any integration that requires an online connection (like checking web emails, fetching web data, etc.). These will be deployed on cloud infrastructure that ensures accessibility from anywhere for the web client.
- **Cloud Hosting Choices:**
- **Application Backend:** We can containerize the backend and deploy on a free tier PaaS. For instance:
  - *Fly.io*: allows a small VM (256MB RAM, etc.) across their global network free for some hours each month. We could run our Node/Python server here. It's good for low-latency globally (if testers are spread out).
  - *Railway.app*: free \$5 credit might cover a month or two of a small Express server with minimal usage.
  - *Deta Space*: would host our code serverlessly with built-in key-value DB if needed (their free tier is generous for hobby projects).
  - *Vercel*: if we go with a Next.js frontend, we can utilize serverless functions for backend. Provided our usage is light, Vercel's free tier (which allows a decent number of function invocations and bandwidth) could suffice. We must watch execution time though (serverless might have limits, e.g., 10s runtime which is okay for quick tasks but not for waiting on the

LLM which might take longer; however, we can have the client poll for the LLM result if needed).

- **Database:** We lean towards using a managed service like *Supabase* (free Postgres). Supabase free tier would cover our 5 users easily with instant setup. Alternatively, if we stick with SQLite, we can host the `.db` file on the same backend server (with proper backups). Given concurrency from only ~5 users, SQLite can manage it, and we ensure writes are minimal. We might start with SQLite (for simplicity) and later migrate to Postgres if needed.
- **LLM Inference Server:** For serving the fine-tuned model, **Hugging Face Spaces** is attractive. We can create a Space (private if needed) that runs a Gradio or FastAPI app which loads the model. On Spaces, if we pick “CPU Basic” hardware (which is free), we get 2 vCPU and 16GB RAM <sup>23</sup>. We will load the 4-bit quantized Mistral 7B (which might use ~8GB RAM) into that environment. The rest of the memory is enough for OS and our service. The drawback is response speed (~1 minute) <sup>31</sup>, but since it’s asynchronous anyway (the user will just get the answer when it’s ready), it’s tolerable for now. Also, 5 users won’t overload it concurrently.
  - We’ll put a note that if this becomes too slow, an upgrade would be to move the model to a service like **RunPod** or **Paperspace** with a GPU. We could schedule the model service to run on a GPU instance only during certain hours (maybe when testers are active) to save cost. E.g., RunPod has an API to spin up containers on demand; we could automate that in a pinch, but probably unnecessary at this small scale.
  - Another idea: Use **HuggingFace Inference API** for some model if available (HuggingFace provides free Inference endpoints for certain models, albeit with rate limits). But since we have custom fine-tunes, running our own Space is fine.
- **Serverless Functions for Automation:** For the scheduled tasks feature, a cloud component needs to execute tasks at certain times (since the user’s machine might be off). We can utilize a free cron service or GitHub Actions scheduled runs to trigger our tasks. For example, we could have a cloud function (on AWS Lambda or GCP Cloud Run) triggered by a cron (these platforms have free quotas for infrequent jobs). The function would wake up, check the DB for any tasks due, then execute them (e.g., fetch emails and store summary for user to retrieve). This way, the user’s device doesn’t need to be always on. Services like **Zapier** or **IFTTT** (free tier) could also be abused to hit a webhook on schedule which then triggers our logic.
- **Feasibility of Hybrid Deployment:** We evaluate that splitting tasks between local and cloud is feasible for user accessibility:
  - Users will get fast response for simple tasks due to on-device handling (no network latency for those).
  - The heavy tasks will take longer due to either API calls or model inference time, but those are inherently slower anywhere. The key is the user interface should handle this gracefully (e.g., show a loading spinner or allow asynchronous responses).
  - If a user is completely offline (no internet), Vezora should still function for local commands and possibly some Q&A via the tiny local model. They obviously can’t use Gemini or fetch online info offline, but basic interactions and controlling the device would still work. This is a big plus of the hybrid approach – partial functionality is retained with no connection.
  - Conversely, if the user is using the **web app** (maybe from a computer that doesn’t have Vezora installed), they won’t have the local modules. In that case, the cloud has to handle more: we might

let the web app use the browser's mic for voice and send audio to the cloud for STT (maybe using Google's free STT API or the Whisper model on our server). App launching features would be limited on pure web (we can only perhaps open web links). So the web version might be a subset of functionality, but memory and conversation are synced. This is acceptable since heavy OS integration likely is only on the installed app.

- **Security of hybrid:** since local app and cloud communicate, we ensure encrypted channels (HTTPS, maybe even an extra layer like a shared secret between each app instance and server). But given few users, we can manage secure tokens.
- **Scaling Considerations:** With 5–6 users, a single instance of each component is fine. If we needed to scale to, say, 50 users, we would need to watch the model server (HF Spaces might not handle many concurrent requests well) and the database throughput. But within our scope, we have a lot of headroom. For instance, Supabase free allows 60 requests/minute – we will be far below that.
- **Testing and Monitoring:** We will monitor resource usage of our cloud parts. Tools like Supabase provide dashboards, and HF Spaces logs usage. We also keep an eye on any free tier limits (like Vercel function invocations). It's unlikely we hit them with so few users, but part of hosting strategy is to ensure we don't accidentally incur charges. If something unexpectedly goes beyond free limits (e.g., one user triggers Gemini too many times), we can enforce quotas in code or cache some responses to cut calls.
- **Geo-Location:** If our test users are all in a region (say all in India given the currency reference), we might choose hosting in a region close to them for better latency. For example, deploying the backend on an Asia data center. Some services allow that even on free (Fly.io does). HF Spaces likely in US/EU, but model inference latency is dominated by compute not network, so that's okay.

In summary, the hosting plan uses **free cloud resources** smartly: a free web host or small container for the backend, free database, free HF Space for the model, and leverages the *user's device as part of the "hosting"* for the real-time interactions. This hybrid deployment not only saves cost but also aligns with user privacy (local processing) and reliability (some tasks don't depend on the network). We also ensure that switching between local and cloud is seamless from the user's perspective. The feasibility has been considered in light of similar architectures (where e.g., an on-device agent handles immediate tasks and defers others to cloud) – this is essentially how voice assistants like Alexa or Google Home operate, though they often rely more on cloud. We tailor that concept to maximize what's done locally. Overall, we can provide a smooth experience without running a single dedicated expensive server 24/7, which keeps us well within budget.

## Step-by-Step Development Plan

To build Vezora in an organized way, we outline a phased development plan. This plan ensures we start with a Minimum Viable Product (MVP) focusing on core functionality, then progressively integrate advanced features, all while keeping security in check and user experience polished.

### Step 1: MVP – Core Chat and Command Execution

**Goal:** Develop a basic version of Vezora that can carry a conversation and execute simple local commands, with one user profile (before multi-user support).

- **Set up the Conversation Loop:** Create a simple chat interface (could be a web page or a console

prototype) that takes user input (text) and displays assistant responses. Integrate the Google Gemini API (or a placeholder echo model first) to handle the AI response generation <sup>43</sup>. Test that we can send a prompt and get a coherent answer. No long-term memory yet aside from what we send in prompt manually.

- **Voice Input/Output Basics:** Integrate microphone input using an existing library or API. For the MVP, we might use the browser's speech recognition (for web Chrome has Web Speech API) or a quick Python speech recognition library for desktop. Likewise, set up text-to-speech output using OS native capabilities (to avoid complexity initially). The goal is to say "Hello" into the mic and hear Vezora respond via speakers.

- **Command Execution Module:** Implement a simple mapping of text commands to system actions. For instance, if user text contains "open Chrome" or "launch VSCode", use Python `subprocess.Popen` or Node `child_process.exec` to run those programs. This can be straightforward string matching in MVP. Test launching a couple of apps via text command.

- **File Operations:** Code a basic file operation through the assistant. Perhaps support a command like "create a file notes.txt" for MVP. This can call OS file APIs. Verify that the assistant can trigger a file creation or opening.

- **Local Memory (Session-only):** Maintain a session conversation history in a list and send it with each prompt to Gemini so it has context. This gives the illusion of short-term memory. At this stage, it's not persistent after the app closes.

- **Testing:** Test the MVP end-to-end: e.g., user says "Hey Vezora, open Calculator", STT converts, assistant launches Calculator, and says "Opened Calculator." Also test a simple Q&A: user asks a factual question, Gemini returns an answer, and it's spoken out. This confirms the core loop: input -> LLM -> output and command execution.

## Step 2: Introduce Multilingual and Voice Enhancements

**Goal:** Expand the assistant's communication abilities to multiple languages and improve speech features.

- **Multilingual Understanding:** Configure Gemini API to detect language or use a language detection library on user input. Ensure that if user speaks or types in, say, Hindi or Spanish, Vezora can respond in that language. Gemini likely handles this automatically, but we test and possibly include a system prompt like "Respond in the user's language."

- **Multilingual TTS:** Integrate or find voices for different languages. For MVP, maybe just English, but here we expand: use Google Translate TTS API or offline voices for a couple of languages to prove it works. For example, use Coqui TTS to generate a Spanish voice if user language is Spanish.

- **Wake Word Implementation:** Implement the always-on wake word detector. Possibly use a small model from Porcupine or Silero, or a simple neural net listening for "Vezora." Integrate it into the app such that the app starts listening only after hearing "Hey Vezora." This is tricky to test, but we can simulate by playing audio or speaking. Optimize it to avoid false positives.

- **Refine Voice Input Accuracy:** Perhaps switch to a better STT (like using Whisper's tiny model for more accurate transcription than default). If using offline STT, test across different accents/languages.

- **User Testing:** At this point, bring in a couple of test users (maybe among the 5-6 target) to try speaking in their native language and performing tasks. Gather feedback especially on recognition accuracy and response correctness.

## Step 3: Expand to Tiered Functionality (Advanced Features)

**Goal:** Add the advanced features like using the fine-tuned local LLM, real-time monitoring, and scheduled tasks.

- **Integrate Fine-Tuned LLM Service:** By now, assume we have fine-tuned Mistral or Phi-2 (done in parallel to coding). Set up the model server (e.g., on HF Space or local for testing). Integrate it into the assistant's logic: develop a decision mechanism for routing queries. For example, implement a function

`answerQuery(user_query)` that decides: if query matches a known simple pattern or the local model is confident, use local model; else call Gemini. We might use a keyword-based approach or even run the query on the small model first to see if it can handle it. This is iterative; test with queries offline vs cloud. Ensure the app can contact this model service (if HF Space, use the API endpoint).

- **Encrypted Long-Term Memory:** Implement the database. At this step, set up the SQLite (or Supabase) and create tables for users, conversations, etc. Implement functions to save conversation turns to the DB as they happen (encrypting content). Also implement retrieval of past context when a new session starts: e.g., load last N interactions to use as context, or allow user to ask “What did I tell you yesterday?” which requires searching memory. Perhaps incorporate a vector store for memory: use a small embedding model (there are open ones like MPNet or MiniLM) to index past conversations for semantic search. This can help in recalling older info without always sending huge context. Test that data indeed is encrypted in DB (we can inspect the file to confirm it’s not plain text).

- **User Accounts & Multi-User:** Introduce the concept of different users. Implement a simple login system (username & password) and tie the encrypted memory to user accounts. For now, maybe just a config file or a prompt at start, but eventually a login screen. Test switching between two users – ensure one user’s data isn’t accessible to the other.

- **Real-Time Monitoring Features:** Build a basic version of an opt-in monitor. For example, implement a thread that checks active window title every X seconds (Windows allows that via APIs). If enabled and if it detects a certain application open or a pattern (like “Outlook” in title), it triggers the assistant to say/print “Do you want me to do XYZ?”. Or use camera: possibly use OpenCV to detect face presence (this could be a simple check if any face-like pattern is present to say “Welcome back!”). Start with one simple monitor (like the active window example) to validate the idea. Provide a toggle in settings to turn this on/off.

- **Scheduled Tasks (Automation):** Implement scheduling using a Python scheduling library (like `schedule` or APScheduler) or cron if in backend. Let the user add a test scheduled task via a command (like “Remind me in 1 minute”). Verify that the reminder triggers at the right time with a notification or voice output. Then expand to daily tasks: integrate with an email API (maybe just Gmail’s IMAP to fetch unseen emails) and schedule it. This requires safely storing email credentials or tokens, which we do encrypted in DB. Test the daily email summary: mark some emails unread, have Vezora summarize them at scheduled time.

- **Cross-Device Sync:** Spin up the web client environment (if not already). Have the web UI connect to the same backend and test that a message typed on web appears in the Windows app if we implement that, or at least that memory persists. This might involve sending push updates or simply verifying that if you ask something on one device, the other knows about it. For now, test sequentially (ask on PC, then later ask on web something related, see if context is there). This ensures our cloud DB and sync logic works.

#### Step 4: Hardening Security and Encrypted Memory System

**Goal:** Before broadening user testing, ensure security measures are robust and the system is stable.

- **Thorough Encryption Test:** Try to manually access the stored data (e.g., open the SQLite file in a text editor or SQLite browser) to confirm it’s encrypted (garbled). Attempt to decrypt with wrong key to ensure that fails. Make sure keys are not logged anywhere. If using Supabase, ensure SSL is on and data looks unintelligible in the dashboard.

- **Penetration Testing (Basic):** Do a pass for vulnerabilities: - Ensure the backend APIs require authentication and check for authorization (one user can’t fetch another user’s data by changing an ID, for instance). - Try some prompt injection attacks on the LLM (like user says “Ignore previous instructions” or tries to get the assistant to reveal internal info). Implement prompt guardrails as needed (for example, always append a system message that refuses to reveal certain secure info). - Check that the assistant doesn’t execute arbitrary commands not intended – for instance, if a user says “delete all files on C:”, do we

have any filter to prevent destructive commands? At least log such attempts and ask for confirmation or simply refuse if dangerous. We can maintain a safe-list of allowable operations in this phase. - Ensure memory DB doesn't grow uncontrolled: maybe set some limits or cleanup policy for old logs if needed. - **Optimize Performance & Resource Use:** Profile the application. Check CPU usage of wake word loop (it should be low). Ensure that the hybrid approach isn't causing undue latency – for example, maybe we need to overlap operations (we can start STT and also load context in parallel to save time). If any step is too slow, consider caching (e.g., cache last LLM answer for identical repeat questions to not call API twice). On the cloud side, see that our HF Space doesn't sleep (Spaces might sleep on no traffic; we might ping it periodically or use the always-on setting if free). - **UX Refinement:** Add loading indicators when waiting for responses, add error handling (if Gemini API fails or internet down, inform the user gracefully: "I'm having trouble reaching my knowledge base, please check your connection."). Make sure the interface is not cluttered and is intuitive. At this stage, possibly involve a UI/UX friend or use Lovable's visual editor to tweak the design.

### Step 5: User-Friendly Deployment & Final Touches

**Goal:** Make the app easy to install/use for our test users and finalize documentation and any tables/reports needed.

- **Package the Windows App:** Use Electron or Tauri bundling to create an installer or standalone application. Ensure that necessary models or files (like the Phi-2 model, or the wake word model) are packaged. Test installing it on a fresh PC (maybe a VM) to see that it runs without the dev environment. Address any missing dependencies. Windows might flag the app since it's unsigned – we may warn users about that since code signing is costly. But within a small test group, that's fine. - **Deploy the Web App:** Push the web client to a hosting (Vercel or Netlify). Test from a phone or different network to ensure it connects to backend. Possibly set up a custom domain (though not necessary for 5 users, a free domain or simply using the provided URL is fine). - **Deployment of Cloud Services:** Make sure the backend is running on chosen host (with auto restart on crash, etc.), the DB is up, and HF Space model is running. If needed, configure uptime monitors (free services like UptimeRobot) to ping these periodically so we know if anything goes down. Given the tiny user count, 100% uptime isn't critical, but we want a smooth demo. - **Final Testing with Users:** Now bring in all 5–6 test users to try out Vezora in their routine. Provide a quickstart guide (how to wake it, example questions). Collect feedback on any issues or desired features. This may lead to minor last-minute tweaks (like adding a particular app integration if someone needs it, or adjusting the assistant's conversational style). - **Documentation & Tables:** Compile documentation for the system – both for users (how to use features) and for devs (how the system is structured, in case of future maintenance). Since the question asks for a *full project report*, we would include in our report some tables summarizing key points, e.g.: - A **feature table** listing free vs advanced features, - A **tech stack table** summarizing each component and choice, - A **cost table** (which we will do in the next section). - **Performance Tuning (if time permits):** If any aspect was underperforming in user tests (say the local model was too slow), consider quick improvements. For example, if HF Space model is too slow, maybe switch to a smaller model for now (Phi-2) for quicker responses at the cost of some quality, or use a shorter context.

By following this step-by-step plan, we ensure that we **build up the system gradually**, verifying each piece before adding the next. This reduces the chance of compounding errors and makes debugging easier. We start with the must-haves (basic chat and execution), then layer on enhancements like multilingual support, then advanced AI features, and finally polish everything. Throughout development, iterative testing with the intended users is key so that by the end, we have a well-rounded personal assistant that meets their expectations.

## Total Cost and Budget Estimate

One of the project's constraints is to keep the total cost under **₹10,000 (approximately \$120 USD)** for development and initial usage by the 5–6 test users. We have carefully chosen tools and services with free tiers or minimal costs to achieve this. Here is a breakdown of expected costs:

Item / Service	Plan / Usage	Estimated Cost
<b>Google Gemini API</b>	Free developer tier (limited requests per day) <sup>6</sup> . Usage by 5 users for Q&A likely within free limits.	₹0 (free)
<b>Open-Source LLM Hosting</b>	Hugging Face Spaces – CPU basic tier <sup>23</sup> for serving Mistral (free). (If needed, occasional RunPod GPU at \$0.4/hour for heavy testing)	₹0 (free)  (~₹80/hour if GPU used, but likely avoid)
<b>Google Colab for Fine-tuning</b>	Colab free tier – used for a few hours to fine-tune models <sup>38</sup> .	₹0 (no cost)
<b>Database</b>	Supabase Postgres free tier (up to 500MB, plenty for text) or SQLite on free hosting.	₹0 (free)
<b>Frontend Hosting (Web app)</b>	Vercel or Netlify free plan for static/front-end. Few users => negligible bandwidth.	₹0 (free)
<b>Backend Server</b>	Free tier on Fly.io/Railway/Deta for a small Node/Python service.  Expect under free quotas given low traffic.	₹0 (free)
<b>Domain Name (optional)</b>	Using free provided domain (e.g., .vercel.app or .deta.dev).  If custom domain needed: ₹800/year (\$10) approx.	₹0 (free)  or ~₹800 if domain chosen
<b>Third-party API integrations</b>	Gmail/Google API – free for personal use within limits (the 5 users using their own accounts).  Other APIs (weather, etc.) similarly have free tiers for low volume.	₹0 (free)
<b>Speech-to-Text API</b>	Using offline STT (Vosk/Whisper) – no cost.  If using cloud STT (like Google Speech), free up to certain minutes per month which we won't exceed.	₹0 (free)
<b>Text-to-Speech</b>	Using offline TTS (Coqui/Mimic3) – no cost.  Alternatively, OS built-in voices (free). If cloud TTS used, small usage under free quota.	₹0 (free)
<b>Development Tools</b>	Lovable and Cursor – They appear to be available in at least trial/free capacities (Lovable may charge for extended use, but building one app might be within free trial; if not, we can do manual coding).  Visual Studio Code and Cursor plugin – Cursor might have a free tier or trial.	₹0 (assuming free usage for dev)

Item / Service	Plan / Usage	Estimated Cost
<b>Infrastructure Contingency</b>	In case free limits are exceeded (unlikely with 5 users), allocate a small amount for upgrades: e.g., \$5 on Railway, \$5 on Supabase (to raise limits), etc.	₹1,650 (\$20) <i>at most</i>
<b>Total Expected Cost</b>	Sum of all the above, expecting mostly free usage.	~₹0 to ₹1,650 (≈ \$0 to \$20)

As shown, the **anticipated cost is well below ₹10,000**. In the best case, we manage to run everything on free tiers, making the out-of-pocket cost nearly zero. In a more realistic scenario, we keep a small buffer (perhaps \$20) for any unexpected needs (for example, if we need to run a GPU instance for a few hours or if Lovable requires a one-month subscription for full use, etc., though we'd try to avoid that).

Some justifications and notes on costs:

- **Google Cloud Credits:** If needed, Google often provides \$300 credit for new accounts – but we likely won't even need to use that for Gemini if we stick to free tier. The usage by 5 users asking a few questions daily is trivial compared to quotas (Gemini's free tier allows thousands of tokens per day) <sup>21</sup>.
- **Model Hosting:** Hugging Face Spaces are free but have one limitation: if the Space is idle, it may sleep, causing a cold-start delay. If that's an issue, we might consider running the model on a small always-on instance. For example, an AWS t3.small (2 vCPU, 2GB RAM) costs around \$17/month, but that's not enough RAM for 7B model. A cheaper way: **Oracle Cloud's free tier** provides 24GB RAM VM always free – that could actually host the model. If one of us qualifies for that, cost remains zero. So we have fallback plans to still avoid cost while ensuring performance.
- **Fine-tuning:** Using Colab saved us having to rent a GPU. Fine-tuning small models only took a few hours, which Colab's free service handled <sup>38</sup>. No paid Colab Pro was necessary.
- **Voice and Data:** The data the assistant might use (like email API calls or fetching news) is all within free developer allowances. For example, Gmail API usage for a handful of calls a day is free. If we integrate any external API, we check their free tier. Many have generous free limits for personal/light use. So no cost there.
- **Development time cost:** Not counted in the ₹10k since it's not an expense, but worth noting we saved dev effort (and by extension cost) using tools like Lovable and Cursor which sped up development cycles.

In conclusion, our careful selection of **free-tier cloud services**, **open-source software**, and **on-device processing** ensures that the project stays **well under the ₹10,000 budget**. Even scaling to a slightly larger test (say 10 users) likely wouldn't incur costs beyond perhaps needing a slightly bigger database (still usually free under hobby tiers). The architecture's efficiency (offloading work to user's device and using small models) minimizes the need for expensive cloud compute. We have also accounted for a small contingency, but it's very likely we won't need the full budget at all – making Vezora not just smart and secure, but also extremely **cost-efficient** to develop and deploy.

---

1 2 16 17 18 19 24 25 26 28 **The Smart Squeeze: Hybrid LLMs with an On-Device NLU Edge - Sensory**  
<https://sensory.com/hybrid-llms-on-device/>

3 4 15 32 33 34 **Lovable vs. Cursor: Which AI Builder Works Better? - Lovable Guides**  
<https://lovable.dev/guides/lovable-vs-cursor>

5 **How can I best secure a local database for a small desktop application? : r/Database**  
[https://www.reddit.com/r/Database/comments/1i4a0jf/how\\_can\\_i\\_best\\_secure\\_a\\_local\\_database\\_for\\_a/](https://www.reddit.com/r/Database/comments/1i4a0jf/how_can_i_best_secure_a_local_database_for_a/)

6 7 21 22 43 **Gemini Developer API pricing | Gemini API | Google AI for Developers**  
<https://ai.google.dev/gemini-api/docs/pricing>

8 9 29 35 **Mistral 7B | Mistral AI**  
<https://mistral.ai/news/announcing-mistral-7b>

10 **Phi-2: The surprising power of small language models - Microsoft Research**  
<https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/>

11 42 **Phi-2 becomes open source (MIT license) : r/LocalLLaMA**  
[https://www.reddit.com/r/LocalLLaMA/comments/18zvxs8/phi2\\_becomes\\_open\\_source\\_mit\\_license/](https://www.reddit.com/r/LocalLLaMA/comments/18zvxs8/phi2_becomes_open_source_mit_license/)

12 13 20 **Hybrid AI Architectures: Merging Cloud Power with On-Premises Security**  
<https://dacodes.com/blog/hybrid-ai-architectures-merging-cloud-power-with-on-premises-security>

14 **Best open source text-to-speech models and how to run them | Blog**  
<https://northflank.com/blog/best-open-source-text-to-speech-models-and-how-to-run-them>

23 31 40 41 **How to Deploy LLM for Free of Cost. | by Incletech Admin | Medium**  
<https://medium.com/@incle/how-to-deploy-llm-for-free-of-cost-6e7947d9b64a>

27 **Create Your Own Local AI Assistant for Enhanced Privacy**  
<https://www.cognativ.com/blogs/post/create-your-own-local-ai-assistant-for-enhanced-privacy/271>

30 **TinyLlama: An Open-Source Small Language Model - arXiv**  
<https://arxiv.org/abs/2401.02385>

36 37 38 39 **Fine-Tune SLMs in Colab for Free : A 4-Bit Approach with Meta Llama 3.2 - DEV Community**  
<https://dev.to/rishabdugar/fine-tune-slms-in-colab-for-free-a-4-bit-approach-with-meta-llama-32-495o>