ChatGPT

Adaptive RAG Chatbot Architecture with Real-Time User Feedback (LLaMA-Based)

Building a Retrieval-Augmented Generation (RAG) chatbot that **continuously learns** from user feedback requires a carefully designed feedback loop and adaptive model update mechanism. Below we present a comprehensive architecture that integrates real-time feedback collection, quality filtering, training data generation, and **adaptive retraining** – all while using **LLaMA** as the base large language model (LLM) and leveraging parameter-efficient tuning (LoRA/adapters) to enable updates on limited hardware. We also describe how this interacts with a LangChain RAG pipeline and ensures ethical robustness. The solution delivers dynamic improvements without full re-training, maintaining responsiveness and accuracy as users interact with the system.

Illustration of a multi-step adaptation pipeline that fine-tunes an LLM with user feedback. The model first undergoes supervised tuning on task-specific data, then collects human feedback from users in deployment, and finally applies reinforcement learning or reward-based fine-tuning to incorporate the feedback [1] [2]*. In our design, we use parameter-efficient techniques (LoRA/adapters) at each tuning stage to minimize computational cost.*

Real-Time Feedback Collection (Explicit & Implicit Signals)

Explicit feedback is gathered through direct user inputs such as rating buttons or feedback forms integrated into the chat UI. For example, the chatbot interface can present a **rating scale (e.g. 1–5 stars or 👍/👎)** at the end of each answer, along with an optional comment box for the user to explain their rating or provide the correct answer if the bot was wrong [3] [4]. A deployment by Amazon collected feedback using a UI where users chose from "strongly disagree" up to "strongly agree" (mapped to scores 1–5) and could additionally supply a better answer or an explanation for dissatisfaction [3]. Such a mechanism ensures we capture not just a numerical score but also **qualitative feedback** on *why* an answer was good or bad. Notably, allowing a short text response for negative ratings is crucial – real-world users often leave only a thumbs-down without context, which skews stats and yields little insight [4]. By encouraging a brief comment (e.g. "The answer missed X detail" or "irrelevant info included"), we obtain valuable signals to drive model improvements.

In parallel, the system monitors **implicit feedback** from user behavior. Implicit cues include cases like the user **editing their query**, rephrasing a question after an unsatisfactory answer, or clicking a "Regenerate"/"Try again" button to request an alternative answer. For instance, if a user asks a question and immediately requests a new answer or re-prompts with a similar question, that indicates the first response was likely unsatisfactory [5]. This action can be logged as a form of negative feedback (even if the user didn't explicitly click a thumbs-down). Likewise, if a user spends a long time on the answer, or copies it, or explicitly says "Thanks, that helps," these could be logged as positive implicit signals. All user interactions with the chatbot are thus instrumented to collect a stream of feedback data points (explicit ratings, textual comments, and implicit usage signals), each tied to the specific question, the context passages retrieved, and the bot's answer.

To implement this, one can embed feedback collection in the application logic (for example, using Streamlit or a web UI). In practice, developers have used simple callback APIs to log feedback. Tools like **Helicone** offer a feedback logging API that attaches feedback metadata (e.g. `rating: true/false` or scores 1–5) to each LLM response ID [6] [7]. The system should assign each Q&A interaction a unique ID and record any feedback with that ID for downstream processing. By **coupling feedback to specific responses**, we maintain a clear mapping between a given model output and the user's evaluation of it [8]. All feedback data is stored in a **feedback database** (this could simply be a table in Postgres, or any scalable logging system) with fields for user ID, conversation/session ID, the prompt, the full response given, references to the retrieved documents, explicit rating (if any), and any textual comment or correction from the user.

Feedback Filtering and Quality Assessment

Collecting feedback is only the first step – the raw feedback must be **filtered and validated** to ensure that we learn from high-quality signals. Not all user input is trustworthy; some feedback may be noisy, biased, or even malicious. The system therefore applies a **feedback quality assessment module** before using this data for model updates.

1. Spam and toxicity filtering: All feedback submissions (especially free-text comments or user-proposed answer corrections) are run through content filters. We leverage the same sort of **LLM guardrails** used for moderation of prompts/responses to screen feedback content. Any feedback containing hateful, profane, or irrelevant content is discarded to prevent the model from learning undesirable behavior. This addresses the Microsoft *Tay* scenario, where an AI chatbot that learned from public tweets was quickly corrupted by troll feedback – a cautionary example that underscores the need to **guard against adversarial inputs** [9] [10]. In our system, if a user attempts to spam the bot with incorrect facts or toxic remarks under the guise of "feedback," these inputs will be identified and filtered out by automated toxicity detectors and rules (with an option for manual review of flagged feedback).

2. User reputation and identity tracking: The architecture assigns each user (or at least each session) a reputation score that gauges their feedback reliability over time. This can start simple – e.g. treat feedback from trusted domain experts or internal testers with higher weight than feedback from anonymous new users. As the system gathers data, it can adjust a user's reputation if their feedback often aligns with others or is later proven correct. Conversely, if a particular user frequently gives divergent or low-quality feedback (or tries to game the system), the weight of their inputs is reduced. For instance, if multiple users disagree with the bot's answer to a certain question but one user says it was great, that lone positive feedback might be down-weighted. The system can implement this by clustering feedback by question and looking at consensus, or by tracking each user's agreement with the majority in past cases. High-reputation users' feedback could bypass some filtering, whereas low-reputation or new users' feedback might require additional validation (or only be used if corroborated by other signals). This **trust scoring** ensures that the training pipeline isn't hijacked by a single bad actor. (In consumer settings, this is analogous to how platforms like StackOverflow trust high-rep users' contributions more.)

3. Bias and error detection: The feedback analysis module also checks for systematic biases. For example, if a user's correction to an answer introduces a fact that contradicts the source documents or known ground truth, the system flags it. We might cross-verify user-proposed answers with the knowledge base: e.g. perform a quick search (using the same vector store or an external API) to see if the correction is supported by any reliable document. If a user claims "the answer is wrong, it should be X," but our verification finds no evidence for X (or finds it false), that feedback is either discarded or queued for human review before trusting it. Additionally, the system monitors feedback for *agenda-driven* inputs – e.g. a user consistently giving negative feedback whenever the bot mentions a certain policy or refusing correct answers due to personal beliefs. Such patterns can be detected through feedback metadata and handled (either by excluding that feedback or by isolating it if we plan to perhaps personalize for that user separately). Overall, **poor-quality content is filtered out and only high-quality, validated feedback is allowed into the training data** [11]. This careful curation mirrors the way base LLM training datasets are highly curated for quality; as one observer noted, *"the dataset used for training needs to be of high quality – if you train a model with poor-quality content, the output will also be poor"* [11]. Our pipeline therefore treats the feedback data with the same rigor as any training corpus, cleaning and vetting it before use.

4. Aggregation and consensus: To increase reliability, the system can aggregate feedback over multiple interactions before acting. Rather than immediately retraining on one user's opinion, it might wait until a certain question or answer has been flagged by **multiple independent users**. For example, if 10 users asked variants of "What is the capital of X?" and 7 of them gave a thumbs-down saying the answer was incorrect, that's a strong signal the model is wrong on that fact. One user's correction might be the outlier, but repeated feedback indicates a genuine issue. By aggregating statistics (e.g. error frequency per question or topic), we prioritize what to learn from. This is analogous to *pattern analysis* on the feedback: identifying common pain points that many users highlight [12]. The system's feedback database can be queried periodically to find trends – e.g. "10% of medical domain questions get low ratings due to lack of detail" or "users often edit the bot's code answers, implying mistakes in code generation." These insights inform which fine-tuning tasks will yield the most impact.

In summary, the feedback quality assessment acts as a **guardrail**, making sure that only **legitimate, representative, and constructive feedback** is used to update the chatbot. This prevents the model from drifting in the wrong direction due to noise or malicious inputs. By combining automated filtering (for toxicity/spam), statistical analysis (for consensus), and reputation systems (for trust weighting), the architecture maintains a high signal-to-noise ratio in the feedback loop [10].

Transforming Feedback into Training Data

After filtering, we have a repository of valuable feedback signals – now we must **translate these into a form the model can learn from**. This involves constructing **pseudo-labeled training data** or reward signals from the raw feedback. There are a few data transformation techniques used in our architecture:

- **Supervised fine-tuning data (QA pairs):** The simplest case is when a user provides the correct answer or an improved answer in their feedback. For example, suppose the user's question was: "What are the steps in the company's onboarding process?" and the bot's answer missed a step. The user responds with a comment: *"The answer is missing the final step, which is scheduling a welcome meeting."* We can convert this into a new training sample: **Input**: (original question, possibly plus retrieved context) → **Target Output**: (user's corrected answer). This is effectively a labeled QA pair sourced from a real interaction. We add this to our **feedback fine-tuning dataset** as a high-quality example of the desired behavior. Over time, collecting many such user corrections yields a dataset of **real-world queries and the answers that humans consider correct**, which is incredibly valuable for tuning the model. Even a simple strategy of "collect a dataset and fine-tune the LLM on it when it's big enough" can be very effective [13] – the key is that the data comes from actual failures/ successes of the deployed model, so fine-tuning on it directly targets the model's weaknesses.

- **Preference labels for RLHF:** Not all feedback comes with an explicit correct answer. Often we have **preference data** – e.g. a thumbs-up or thumbs-down on the model's output. We can leverage this for **reinforcement learning** or preference modeling. One approach is to create **comparison samples**: for instance, if the user asked a question and we have one answer that got a thumbs-down and perhaps a later improved answer that got a thumbs-up, we can form a preference pair. In reinforcement learning from human feedback (RLHF), training often uses tuples like [Prompt, Answer_A, Answer_B] with a label of which answer was preferred [14]. We can similarly construct such pairs from our logs: Answer_A = the original disliked output, Answer_B = either a corrected output (if available) or the output after some refinement, and the feedback indicates which was better. Even if we don't have a second answer from the model, we could generate one (for instance, have the model produce an alternative answer after a user downvote, or use an expert-written answer) and label it as the "chosen" versus the original as "rejected." These comparisons feed into training a **reward model** that learns to score outputs, which can then be used to fine-tune the policy (the chatbot model) via RL to prefer outputs with higher reward [15]. In practice, a large volume of such preference data is needed for stable RLHF, so we might combine our user feedback with additional synthetic preferences. (As a hybrid approach, Amazon's team augmented limited human feedback with AI-generated feedback to scale up training data for RLHF [16] [17], a technique we could employ if needed.)

- **Categorizing and segmenting feedback for targeted training:** We don't necessarily fine-tune on all feedback at once; often it helps to **segment the feedback by category** and create specialized fine-tuning sets for different objectives [18]. For example, cluster the feedback data into: factual corrections, formatting/style improvements, reasoning chain improvements, etc. If users often say the answers are too verbose, that's a stylistic issue – we can assemble those instances and fine-tune the model to be more concise for those prompts. If a subset of feedback is from a particular domain (say, legal questions) showing the model's answers lack detail, we can fine-tune the model on high-quality Q&A from that domain (possibly including user-supplied answers or additional ground-truth from documentation). In essence, we **analyze the feedback to identify common issues and success cases** [12], then curate focused datasets to address each. For instance, we might create a "hallucination correction" dataset containing all instances where the model's answer was factually wrong along with the true answer (sourced from users or from a trusted knowledge base). Training on this will directly reduce future hallucinations. By organizing feedback this way, we ensure each fine-tuning round has a clear objective and doesn't inadvertently skew the model in unrelated areas.

- **Data augmentation and validation:** Before using feedback-generated data, especially user-written answers, we may do a sanity check or augmentation. One technique is to have the model (or another model) **double-check the user's answer** against available references. For example, use the retrieval system to fetch documents relevant to the question, and verify that the user-proposed answer is supported. If it is, we can mark that feedback example as verified true. If not, we might either discard it or try to **complete/correct** the answer using an automated approach. A possible automated approach is to use an LLM to act as a "teacher": provide it the question and the documents that should contain the answer, and ask it to write the ideal answer. This AI-generated answer can serve as a reference. In fact, a study by Google DeepMind (April 2024) found that aligning LLM outputs with user feedback (through such iterative refinement) led to a significant increase in positive user interactions [19]. That is, if we consistently incorporate user feedback and verified corrections into training, users will notice the bot getting better – leading to higher satisfaction in a virtuous cycle.

Once we have curated and segmented the feedback-informed data, we **export these as training datasets** (e.g. in JSON or CSV with prompt/response pairs or comparison tuples) [20]. At this point, we have effectively turned raw user feedback into **actionable training data**. Each fine-tuning iteration will draw on these datasets. The pipeline can maintain multiple sets: one for supervised fine-tuning (SFT) and one for RLHF, etc., depending on the training approach used in the next stage.

To summarize this stage, the system **filters and transforms user feedback into training signals** through a combination of supervised targets and reward-based labels. Real user interactions yield a specialized dataset of real-world Q&As and preferences, which can be exported periodically for model updating [20]. This ensures our model update process is driven by actual usage patterns and user-defined correct behavior, rather than hypothetical data. By systematically curating this "experience replay" from the chatbot's conversations, we lay the groundwork for continuous learning.

Efficient Fine-Tuning with LoRA/Adapters (Selective Model Updates)

With fresh training data in hand, the next component is the **adaptive retraining mechanism** – updating the LLaMA-based model using the feedback-derived data. Fine-tuning a large model from scratch for every update would be computationally prohibitive, so our architecture uses **parameter-efficient fine-tuning** techniques, namely **LoRA (Low-Rank Adaptation)** or adapter layers, to incorporate updates cheaply and quickly.

LoRA Adapters: LoRA introduces small trainable weight matrices (of low rank) that are inserted into the model's layers (for example, it can add low-rank update matrices to the query and value projection matrices in each transformer block). During fine-tuning, the **base model's weights are frozen**, and only these small adapter weights are trained. This drastically reduces the number of trainable parameters (often by a factor of 100x or more) while still allowing the model to learn the necessary adjustments [21] [22]. For instance, in one implementation, fine-tuning a 7B model with LoRA required updating only ~5.7% of the parameters to adapt to a domain [23]. These LoRA weights are then applied on top of the base model weights at inference time, modifying the model's behavior. The key benefit is that the adapter weights are **small** (typically a few megabytes), and fine-tuning them requires much less memory and compute than full model tuning. In fact, by combining LoRA with 4-bit quantization of the model (a technique known as **QLoRA**), researchers have shown a **90% reduction in memory footprint** – fine-tuning LLaMA-2 7B could be done in 9–14 GB of GPU memory instead of hundreds of GB [24]. This means even a single consumer-grade GPU (such as 16 GB or 24 GB VRAM) can handle the fine-tuning process, making continuous updates feasible on local infrastructure. *"Techniques like PEFT, specifically QLoRA, can significantly reduce memory footprint by up to 90%… training only 0.1–1% of parameters"* [21] – allowing us to adapt LLaMA on modest hardware.

In our architecture, we maintain the **base LLaMA model** as is (this could be LLaMA-2 Chat model fine-tuned initially on some domain data or instructions) and **stack one or more LoRA adapter modules** on it for feedback-based tuning. Each time we retrain on new feedback data, we are effectively adjusting the adapter weights. We can either *update the same adapter weights continually* (cumulative learning) or use a sequence of adapters – for example, load a fresh LoRA module for each training batch. There are advanced strategies like merging LoRA weights or **S- LoRA** (sequential LoRA) where multiple adapter deltas can be combined over time [25], but a simpler approach is to have one adapter that we update progressively. Because the LoRA modules are small, we could even maintain **multiple specialized adapters** (e.g. one for a certain domain or one for a certain style) and toggle or merge them as needed. An engineer on an online forum suggested a similar concept: training LoRA adapters from conversation data and even merging different

LoRAs over time, using them alongside retrieval to avoid forgetting base knowledge [25]. This way, the system could learn new user preferences via LoRA while relying on RAG to supply factual knowledge, striking a balance between learning and memory.

Adapter layers: As an alternative to LoRA, one can insert **adapter layers** (small feed-forward networks) at each transformer block. These serve a similar purpose: they introduce a few million trainable parameters that can be fine-tuned, leaving the main model unchanged. At inference, they are simply extra computations per layer. Adapters and LoRA both allow **selective fine-tuning** – crucially, the original model's weights (which encode general language ability and prior knowledge) remain intact, reducing the risk of catastrophic forgetting. We are only "dialing in" new behavior by **adding tiny weight updates.** This is particularly important for a RAG system: we want to improve the model's ability to use retrieved knowledge and follow user preferences, *without overwriting its fundamental training or exhausting its capacity with new facts*. Research community insights indicate that LoRA-type adapters are great for "**improving the effective use of RAG** rather than storing lots of new knowledge" [25] – in other words, the retrieval system provides the latest info, and the adapter helps the model integrate that info correctly and follow preferred styles or guidelines.

Fine-Tuning Process: The retraining mechanism can operate on a **scheduled cycle** (e.g. nightly or weekly fine-tuning jobs) or trigger in **real-time** once a threshold of new feedback data is reached. For example, if by the end of the day we have 100 new high-quality Q&A pairs from feedback, we launch a fine-tuning run on those. Thanks to LoRA, this fine-tuning might take only a few minutes to an hour on a single GPU for a 7B-13B model (depending on the batch size and epochs). Empirically, small batches of domain-specific data often converge quickly – as reported in one use-case, ~137 samples fine-tuned with LoRA for 20 epochs were enough to significantly improve a model's domain performance [26]. We similarly expect that each incremental training on recent feedback will make **small, cumulative improvements** to the chatbot. The fine-tuning job would load the latest base model + current adapter weights, apply further training on the new dataset, and then save the updated adapter. We keep the learning rate low and potentially limit epochs to avoid overfitting the model to just the recent interactions (preventing it from skewing too far due to a few examples).

Once the adapter is updated, the **deployed model is switched** to use the new adapter weights. This can be done by hot-swapping the weights in memory if the framework supports it, or by reloading the model with the new adapter (which is typically a fast operation since the base model is already in memory, we're just loading a few MB of diff weights). Because we are not altering the base weights, these updates can be rolled out with minimal downtime – for instance, one can maintain two instances of the model and rotate them: one is serving while the other updates, then switch. The net effect is that **the chatbot improves its responses over time**: questions that it previously answered poorly (and for which it received negative feedback) will gradually be handled better after the targeted fine-tuning. A study has noted that such real-time fine-tuning combined with RAG can reduce answer variability and improve accuracy as the model adapts to new data and feedback patterns on the fly [27]. Our approach aligns with that insight: by continuously training on new conversation data, we help the model "lock in" corrections and preferred behaviors, yielding more consistent results.

LoRA for RLHF: It's worth noting that even if we employ a reinforcement learning stage (PPO training with a reward model), we can still use LoRA to reduce costs. In an AWS implementation, the team used PEFT/LoRA during the PPO fine-tuning of a 7B model (with LoRA rank=32) to accelerate RLHF updates [28]. We can do similarly: the policy model (LLaMA chatbot) gets additional LoRA gradients from the RL objective. This again keeps the computational burden low and allows running an RL update loop on a single machine if needed. However, RLHF is more complex and volatile [15], so in many real-world cases, a simpler **supervised fine-tuning on curated feedback data** (possibly with some reward-model filtering of which outputs were good/bad) may suffice to get big gains in alignment and factual accuracy.

In summary, using LoRA/adapters lets us **inject knowledge and preference updates into the model frequently, without full retraining**. We preserve the efficiency of the base LLaMA model (no need to re-compute all its weights) and simply compose the learned "deltas" on top. This design is inherently suited for **local deployment on limited GPUs** – as an example, one could fine-tune a LLaMA-2 13B on a single 24 GB GPU with QLoRA [21], and produce an updated adapter that is loaded into the running chatbot, all within a short time window. By planning these updates at regular intervals, the chatbot essentially **evolves** with usage: it's a step towards "online learning" in a controlled manner. Multiple experts anticipate that continuously training LLMs through user interactions will become increasingly viable with such techniques [29] – our architecture embodies this by combining RAG (which handles unseen knowledge) with lightweight fine-tuning (which handles learned preferences and fixes).

Importantly, we ensure that the **core model remains stable**. If anything goes wrong with an update (e.g. a buggy fine-tune that teaches the model a bad habit), we can revert by disabling the latest adapter. The base LLaMA model (possibly with an initial alignment fine-tune) acts as an anchor. The LoRA updates are additive and can be reset or adjusted without permanently altering the foundation model. This modularity adds a layer of safety and flexibility to our continuous learning approach.

Dynamic Retrieval & Knowledge Base Updates (PGVector Integration)

While the model adapts to user preferences and common mistakes via fine-tuning, an equally important part of the system is the **retrieval component**. Our chatbot uses a RAG approach: it relies on a vector database (PGVector) to fetch relevant knowledge for each query. Ensuring this knowledge source stays up-to-date and **improves over time** is crucial. Thus, the architecture includes a mechanism for **dynamic retrieval index updates** based on feedback.

1. Ingestion of new information: Often, user interactions will highlight gaps in the knowledge base. For example, if the bot responded "I don't have information on that" or gave an outdated answer, the user might provide the latest info in their feedback ("Actually, the policy was updated last month – here is the new rule…"). In the AWS pilot, some users even supplied the document excerpt that contained the correct answer when the model hallucinated [30]. Our system capitalizes on this by immediately ingesting such **user-provided references or new data**. If a user links to a URL or uploads a file to prove the answer, an ingestion pipeline takes that content, chunks and embeds it (using the same embedding model as the rest of the KB), and **upserts it into the PGVector index**. This means that for the very next user query on that topic, the retrieval component can surface the newly added fact. Dynamic updates to PGVector are relatively straightforward – since PGVector is built on PostgreSQL, inserting or updating embedding records can be done with an SQL UPSERT, and the index (IVFFlat or HNSW) can be refreshed as needed. We could even maintain a real-time update process: e.g. after each feedback, push new vectors and **recompute any indexing structure** in the background. Because the volumes are not huge (most enterprise chatbots have a knowledge base on the order of thousands to millions of documents, where single document additions are fine), this is feasible.

2. Removing or flagging faulty data: Conversely, if feedback reveals that certain content in the vector store is misleading or low-quality, we can flag it. For example, if the bot consistently retrieves a particular document chunk that leads to wrong answers (maybe a piece of outdated documentation), and users keep indicating those answers are wrong, the system can respond by lowering that chunk's priority or removing it from the index. One strategy is to attach a **score penalty or tag** to vector entries that are associated with bad answers. The retrieval component could be adjusted to incorporate such a score: e.g. we store a field `quality_score` with each document embedding in PGVector, and adjust the similarity ranking by this score (this might require a custom retrieval query or a reranker that filters out low-quality hits). A commenter described it as introducing a "multiplier or ranking system on the vectors to promote higher relevancy or degrade others" based on feedback [31]. For instance, if "Document A" was used in three answered questions that all got a thumbs-down, the system could learn to **trust Document A less** – either by not retrieving it unless it's a top match or by demoting its similarity score a bit. This is a form of **reinforcement learning on the retriever**. While it mainly affects identical prompts (since retrieval is query-dependent), over time, maintaining a curated set of documents ensures the model sees more reliable context.

3. Curating a growing Q&A knowledge base: As users ask new questions, we can accumulate a set of Q&A pairs that have been verified. These can be turned into an **FAQ-like knowledge base** which is also indexed in the vector store. One suggestion from a community member was to "group the best answers by topic and store them in your vector DB" so that they appear as candidate context for future queries [32]. We implement this by taking answers that got very positive feedback (high ratings) – presumably, these are high-quality answers, possibly user-corrected or carefully generated – and storing them as reference texts. For example, if the bot now has a perfect answer about "How to reset my password" after incorporating feedback, we save that answer (or the underlying official answer from documentation) into a "Q&A reference" collection in PGVector, labeled with the topic. Next time a similar question comes, the retriever might pull in this reference answer, further improving consistency. Essentially, the chatbot can start to **leverage its own past validated answers** as part of the context. This yields a self-reinforcing effect: good answers beget more good answers. Of course, we must ensure these Q&A entries are kept updated (if the domain information changes, update the entry as well).

4. Updating embeddings or retriever logic: Another aspect is the embedding model itself. Over a longer term, if we find that certain queries are not retrieving relevant documents even though they exist in the knowledge base, it might indicate the embedding model isn't well-suited to those types of questions. Our architecture allows for periodically fine-tuning or replacing the **embedding model** (used by PGVector) as well. For example, we might fine-tune a SentenceTransformer on our domain's Q&A pairs so that it better maps questions to relevant docs. This can be done offline and then the new embeddings are computed for all documents (or gradually updated). While this is not real-time per user, it's part of the adaptive cycle to ensure retrieval quality improves. The research paper we saw recommends *"finetuning the embedding model"* and exploring advanced retrieval methods to improve context relevance [33] – reflecting that a smarter retriever reduces the burden on the generator and yields more accurate answers. In a LangChain context, one could also incorporate a **reranker LLM** that looks at the retrieved docs and the query, and sorts or filters them. This reranker itself could be updated using feedback (train it to give higher scores to documents that lead to good answers).

Integration with PGVector: PGVector offers the convenience of using standard SQL for managing the vector store. We design a **knowledge base service** that listens for new content or feedback events. When a new piece of content is submitted (via user feedback or via an upstream data source update), this service computes its embedding (using our embedding model, possibly hosted or via an API) and then issues an `INSERT` into the `vectors` table (with embedding and metadata). Similarly, when content is deprecated or known-bad, it can either delete that row or mark a flag in metadata. The retrieval query (for example, using `pgvector`'s `<->` operator for cosine distance) can be augmented to exclude flagged items. The nice part is that these operations can happen **concurrently with chatbot usage** – PGVector is essentially a live index, so writes can be done without downtime. We do need to ensure the query performance remains good (very frequent writes might degrade the index search temporarily), but given typical usage, the volume of updates is low relative to read queries.

Finally, we maintain an **embedding of user feedback itself**. That is, in addition to updating the knowledge base, we could store an "embedding of the feedback" or a record linking the question to the corrected answer. This could help in meta-reasoning. For instance, if a very similar question comes in future, the system could recognize that "this looks like that other question where the user didn't like the answer" and proactively avoid the previous mistake. In LangChain, one might implement this by storing the conversation history in a vector store as well (conversational memory). Our architecture can incorporate a **conversational memory vector store** for the model to retrieve relevant past dialogues. This is separate from the main knowledge base, but it's another dynamic data source that grows with feedback. It ensures the bot not only retrieves static facts but can also *retrieve lessons learned from prior interactions*.

In summary, the retrieval component is **continuously refined**: new knowledge is added as it emerges (or as users supply it), and problematic or stale knowledge is removed or down-weighted. The combination of these actions means the RAG pipeline stays current and accurate. By the time we also fine-tune the model on the corrected answers, the whole system (retriever + generator) is improved: the retriever offers better context and the generator is better at using it. This two-pronged learning (updating both knowledge and the model) is what enables robust real-time adaptation.

Integration with the LangChain-Based RAG Pipeline

All the above components – the LLaMA model (with LoRA adapters), the PGVector vector store, and the feedback loops – need to work in concert. We utilize **LangChain** as the framework to orchestrate the RAG workflow and incorporate the feedback mechanism into the chatbot application.

RAG Pipeline Overview: The core question-answering flow follows a typical LangChain RetrievalQA (or ConversationalRetrievalChain) pattern: 1. **User query -> Retriever:** The user's question (along with conversational context, if any) is passed to a LangChain retriever object. This uses the PGVector store to find relevant documents or snippets. For instance, LangChain's `VectorStoreRetriever` can be configured to query the PGVector-backed vector store with the question embedding and return the top-k similar text chunks. 2. **LLM generation:** The retrieved snippets plus the user's question are then fed into the LLaMA LLM (the generation step). LangChain helps format the prompt, for example: *"Given the following context, answer the question..."* and appends the relevant docs. LLaMA (with the latest fine-tuned weights via LoRA) produces the answer. 3. **Output to user:** The answer is displayed to the user in the chat UI, along with citations if we include source attribution. This is the response that the user can then evaluate and give feedback on.

LangChain makes it straightforward to implement this sequence as a chain. We deploy LLaMA, for example, via the Hugging Face transformers integration or a local endpoint, and wrap it in a LangChain `LLM` object.

The vector store PGVector is wrapped with LangChain's `FAISS` or `PGVector` integration (LangChain supports PGVector as a vector store). In fact, an example pipeline from a blog used Llama-2 with a vector store and LangChain to handle retrieval/generation, demonstrating the viability of this stack [34]. Our system follows that proven template.

Integrating feedback collection: LangChain provides callback hooks and the LangSmith platform for tracing and feedback. We utilize these to hook into the moment after the LLM produces an answer. A custom **callback** captures the prompt, retrieved docs, and final answer, and waits for the user's explicit feedback. If using LangChain's agent or chain infrastructure, one could incorporate a step where after outputting the answer, the chain expects a "feedback input" from the user (this can be out-of-band via the UI rather than a chain link). Practically, the feedback (rating/comment) can be sent to a feedback API (like Helicone's) or directly to our backend which logs it. We can also store the entire interaction trace via LangSmith: the question, context, answer, and user feedback can be saved as a **dataset run**. LangSmith's dataset features would allow us later to query, filter, and export this data for training [35].

The RAG pipeline is extended with a **feedback loop sub-pipeline**: - After an answer is given, the system *renders feedback controls* (the UI elements for thumbs up/down, etc.). - When feedback is submitted, it triggers a **FeedbackHandler** component (this could be a Python function in the app or a serverless function) which performs the steps described earlier: filter the feedback, store it in the database, possibly update the vector store if a new document is provided, and add it to the training data queue. - Optionally, we can use the feedback immediately to influence the next response in the same session. For example, if the user clicked "dislike" and requested a new answer, the chain could adjust: maybe re-run the retriever with some variation (e.g. use the user's comment as an additional query). However, this kind of instant adjustment is more of a heuristic fallback (like simply try searching different keywords). The true learning will come in subsequent interactions after model retraining. That said, an advanced feature could be a **self-reflection** step in the chain: if the user flags an answer as bad, the agent (using a system prompt) could analyze why and attempt a corrected answer on the fly. LangChain's flexibility allows constructing such chains (akin to "self-correction" prompts or using another tool when feedback is negative). For example, a *Self-Reflective RAG* approach might prompt the LLM to critique its prior answer if the user was unhappy [36]. This can provide a better immediate answer while the longer-term learning catches up.

Continuous learning pipeline: We envision the whole system as a **workflow pipeline** where LangChain handles the request/response and logging, and a background process (or separate service) handles the training updates. Concretely: - The *frontend* (LangChain chain + UI) is concerned with answering queries and collecting feedback. - The *backend trainer* (could be a scheduled job or event-driven function) monitors the feedback database. If it sees, say, N new feedback items or it's a scheduled time (e.g. 2 AM daily), it will fetch the new feedback data, perform the transformation (prepare training batch), run the fine-tuning (with libraries like Hugging Face Transformers + PEFT for LoRA), then update the model weights in the serving infrastructure. - The *retriever updater* might be integrated in the same backend or separately, which updates PGVector with new content or flags as discussed.

Using LangChain in the loop doesn't require changes to LangChain itself – rather, we **embed our adaptation logic around it**. It provides the structured interface to the model and vector DB, and we plug in our custom components (feedback DB, trainer, etc.). The modularity ensures we can swap pieces (for instance, if we move from PGVector to another vector store, or change the LLM model) with minimal changes to the chain logic.

Finally, by leveraging LangChain's observability and possibly test suite, we can keep track of how the model is improving. LangChain's evaluation tools (and LangSmith) can periodically run a set of test queries through the chain and compare outputs over time. We would expect to see, for example, the score for answers to previously problematic queries improve after the learning loop. This gives the technical team feedback on the feedback loop itself, closing the meta-loop of monitoring performance.

In essence, **LangChain acts as the glue** that ties the LLM, the vector store, and the feedback loop into a coherent application. It enables **rapid development of the prototype** and provides the means to scale to production (with features like caching, streaming, etc., which can be used in our deployment as needed). The architecture we propose is well-aligned with LangChain's design philosophy: keeping the LLM reasoning grounded in external data (via RAG) and continually enhancing the system through feedback and evaluations.

Ethical Safeguards and Robustness to Adversarial Feedback

Adapting a model in real-time based on user input raises important **ethical and safety considerations**. We incorporate multiple safeguards to ensure the chatbot remains aligned with ethical guidelines and is robust against manipulation or noise in the feedback.

1. Maintain a base aligned model: We start with an **aligned LLaMA model** (for example, LLaMA-2 fine-tuned with instruction-following and harmlessness via RLHF). This base model has undergone extensive training to avoid toxic or biased outputs. In our adaptive setup, we **never overwrite or forget this alignment training**. The LoRA fine-tuning is constrained to specific tasks (answer correctness, style improvements) and we explicitly *exclude* any feedback data that would push the model to unethical behaviors. For instance, if a malicious user tries to give feedback like "You should respond with insults next time" or attempts to teach the model toxic language, our filtering would catch and reject it. By keeping the base model fixed and only adding small changes, we reduce the risk of the model drifting into unsafe territory. The adapter updates are predominantly on knowledge utilization layers, not on fundamental language or values layers.

2. Feedback moderation: As discussed in filtering, we perform **content moderation** on user feedback. Any feedback that contains hate speech, personal data, or instructions to break the rules is not used for training. This is analogous to how one would filter the training data of the original model – we continue that hygiene practice in the feedback loop. Moreover, we do not allow a single user to immediately change the model's behavior. All updates are done on aggregated, vetted data and typically require multiple instances or additional validation. This prevents an attack where an adversary might flood the system with a particular viewpoint in feedback to skew the model. The system can detect unusual spikes or repeated feedback from one source and flag them. In an extreme case, if we suspect a coordinated attack (say a group of users all providing the same incorrect feedback to push the model towards disinformation), we would ideally have a human administrator review those before any model update is applied.

3. Guardrails on the learning process: We incorporate **automated evaluation checks** before deploying a new model version. For example, after fine-tuning on feedback, we run a set of safety tests (a suite of prompts known to test the model's boundaries, such as prompts about self-harm, violence, etc.). If the new model version exhibits any regression on those safety tests (e.g. it now starts giving advice it shouldn't, or it became more biased), we rollback the changes and investigate the training data to filter any problematic influence. This is important because even innocuous feedback could unintentionally bias the model. For instance, if users from a particular demographic give a lot of similar feedback, the model might become skewed to that perspective. Our evaluation would try to catch such shifts. We also use **zero-feedback baselines** as control: the base model outputs are periodically compared to the adapted model outputs for a random set of questions, to ensure that except for the intended improvements, the adapted model hasn't lost generality or picked up unsafe tendencies.

4. Rate of adaptation and update frequency: To maintain stability, we typically won't update the model on a per-interaction basis (especially not for ethically sensitive aspects). Real-time in this context means *prompt* (perhaps daily or weekly) adaptation, not instantaneous. This allows time for oversight. It also ensures that any single user's bad feedback doesn't immediately ruin the model for others [10]. The continuous learning is done in *small increments*, which is easier to monitor than one big end-to-end retraining. If anything starts going wrong (we notice a slight increase in an unwanted style), we can correct course in the next batch.

5. Transparency and user consent: From an ethical standpoint, users should be informed that the system learns from their feedback. This can be communicated in the UI (e.g. "Your feedback will be used to improve our AI"). Optionally, give users control over whether their specific conversations can be used for training (privacy concerns). For instance, if the chatbot is used internally in a company, users might flag certain conversations as sensitive (not to be included in training data). We respect these flags and only use feedback from conversations that are allowed. All training data derived from user interactions should be anonymized and stripped of personal identifiers to protect privacy.

6. Adversarial testing: We will conduct adversarial tests on the feedback loop itself. For example, have red-team users attempt to poison the model via feedback – our system's reputation and filtering mechanism should withstand such attempts. Additionally, we guard against **model self-feedback loops**. We do not blindly trust an LLM to generate its own training data without human oversight, as that could lead to a feedback spiral of errors. (We do use an LLM for AI feedback scoring in a controlled way, as mentioned, but even that is supervised by humans in our design [17] [37].)

7. Preserve factual grounding: A potential risk with learning from user feedback is that the model might learn *from the users' mistakes or opinions*, which could be incorrect. To counter this, we always tie the model's learning to the ground truth sources. If a user's feedback says the answer was wrong, we ensure we find the **true answer in the documentation** and use that as the training target, rather than the model just learning "don't answer that question". By training on "question -> correct answer with sources", the model stays factual. Also, when updating the vector store, we only add content from authoritative sources (or clearly mark user-provided content as such). The model can be instructed to mention if a piece of info is user-provided and unverified. These practices keep a clear line between verified knowledge and user opinion, reducing the chance of the model accumulating false info. As one commentator noted: fully online learning from arbitrary users can lead to the model "using bad information in its database for everyone" [10] – our design avoids that by rigorous verification steps.

8. Ethical fine-tuning objectives: In the feedback fine-tuning data, we also include any **alignment-relevant feedback**. For instance, if a user flags that an answer was potentially biased or offensive, we treat that seriously and include it in training as a bad example (with the corrected, respectful answer as the target). The continuous learning should thus encompass not just factual correctness but also the tone and politeness. We maintain a special set of "don't do this" examples from feedback if any, to remind the model of the boundaries. Since our LoRA updates are small, they can be targeted to fix specific bad behaviors (like if the model made a joke that was in poor taste and a user complained, we can have a fine-tuning example that the correct response is to apologize or respond solemnly in that context). These act as on-the-fly alignment updates, complementing the base model's general RLHF alignment.

9. Human oversight for critical updates: For domains with high stakes (medical, legal advice, etc.), we might include a *human-in-the-loop* for feedback processing. That is, before the system incorporates feedback about a critical piece of information, a domain expert reviews it. This ensures that the model doesn't learn something incorrect that could have serious consequences. As the AWS case study highlighted, subject matter experts (SMEs) can supervise the AI-generated feedback process to ensure quality [17]. In our case, an SME could periodically review the log of model changes derived from feedback. However, to keep things scalable, the system should be robust enough that such intervention is rarely needed, focusing on automation with high-quality filtering as described.

By implementing these safeguards, we strive to make the chatbot's learning loop **both responsive and responsible**. The system is designed to be **resilient to noisy or malicious inputs**, taking in the wisdom of the crowd while weeding out the "Tay-like" pitfalls of naive online learning [9] [38]. Every adaptation is done with a lens of maintaining alignment, and continuous monitoring is in place to catch any drift. This ensures that as the bot gets smarter and more customized to user needs, it remains **ethical, unbiased, and safe** for all users.

Implementation Strategy: Technology Stack, Deployment, and Update Cycle

Having outlined the architecture, we now detail a practical implementation strategy, including the tech stack components, how to deploy on limited GPU resources, and how to manage the update frequency and latency for real-time adaptation.

Technology Stack:

- **LLM Model:** We use **LLaMA** as the base model, specifically a variant like LLaMA-2 13B-chat or 7B-chat (depending on hardware constraints). The chat-oriented version is preferable for a chatbot, as it comes with initial instruction-following tuning. This model will be hosted locally. We can use Hugging Face Transformers to load the model in 16-bit or 8-bit precision. During inference, libraries like **bitsandbytes** allow running models in 8-bit or 4-bit mode to save memory (for example, a 13B model can run in around 13–14 GB with 4-bit compression). The model will be loaded with options to **inject LoRA adapters** via the Hugging Face PEFT (Parameter-Efficient Fine-Tuning) library. We will maintain one set of LoRA weights that can be applied to the model. Hugging Face's `PeftModel` class makes it trivial to apply LoRA weights on top of a base model for inference.

- **LoRA/Adapter Tools:** For training the model on feedback data, we employ the **Hugging Face PEFT** library which supports LoRA and other adapter methods. If memory is a bottleneck, we specifically use **QLoRA**: quantize the model to 4-bit for fine-tuning, which, as noted, can reduce memory usage dramatically [24]. The training itself can be done with standard PyTorch or Accelerate, possibly mixed-precision (fp16) to further save memory. We also consider **DeepSpeed** or **FlexGen** for optimizing training if needed, but given the small batch sizes and LoRA, plain PyTorch might suffice. If we needed to incorporate reinforcement learning, the **TRL (Transformer Reinforcement Learning)** library from Hugging Face could be used to implement PPO with our reward model – they also support PEFT to train the policy model with LoRA, as evidenced by AWS's use-case [15]. The reward model would be a smaller model like DistilRoBERTa (as AWS did) fine-tuned on preference data [14], but this is an optional component.

- **Vector Database: PGVector (Postgres + pgvector extension)** is the choice for our vector store. This allows us to use a single PostgreSQL instance for both vector data and metadata (and even for feedback data storage, simplifying the stack). We configure PGVector with appropriate indexes (likely an IVF index for large scale, or HNSW if supported) for fast approximate similarity search. The document embeddings can be computed by a **SentenceTransformer** model (e.g. all-MiniLM-L12-v2 as used in some examples [39], or a domain-specific embedding model). Alternatively, we could use the LLM itself to embed (via its embedding API if available), but using a smaller dedicated model is faster and avoids hogging the main LLM for embedding tasks. The vector store will store each chunk of text along with metadata like source document, embedding, and perhaps feedback quality score.

- **LangChain and Orchestration:** We use **LangChain (Python)** to build the chatbot chain. LangChain will interface with the LLM (via HuggingFaceHub or a local pipeline wrapper), and with PGVector (via the PGVector vector store class). We will likely write a custom retriever if needed to incorporate feedback-based filtering. LangChain's `ConversationalRetrievalChain` can manage context (previous chat history) and retrieval for a QA chatbot nicely. We also use **LangSmith** (the LangChain observability toolkit) or an analytics tool to log interactions. This will help in evaluating improvements and debugging.

- **Frontend/UI:** Depending on the deployment scenario, the UI could be a web app (Streamlit was mentioned in some cases [40], but for production a React or custom web interface might be used). The UI needs to support showing retrieved sources (for transparency) and capturing feedback inputs. If using Streamlit for quick setup, it's straightforward to add buttons for feedback [41]. For a more scalable setup, a front-end calls a backend API (say a FastAPI or Flask server) which runs the LangChain chain and returns the answer. The front-end then sends a feedback API call with the conversation ID and rating/comment.

- **Feedback Data Store:** We can use a simple **PostgreSQL** schema for this (since we already have Postgres for vectors). For example, a table `Feedback` with columns: `id, user_id, question, answer, rating, comment, timestamp, resolved_flag`. Alternatively, use a vectorstore's metadata to attach feedback counts to documents (but per interaction feedback is better in a separate table). If we want to track user reputation, we might have a `UserStats` table accumulating things like number of good/bad feedback provided, agreement rate with others, etc.

- **Deployment environment:** The entire system can be containerized with Docker. One container runs the Postgres + PGVector database. Another container runs the Python backend with the LLaMA model. We will need to ensure this container has access to a GPU (using nvidia-docker runtime). If we have only one GPU, that GPU will handle both inference and occasional fine-tuning jobs. If we have two GPUs, we could dedicate one for inference and one for training to avoid contention. LLaMA 7B or 13B with 4-bit quantization can handle a single inference on a 16GB GPU with low latency (~1-2 seconds for moderate output length). Fine-tuning such a model on a small batch could take a few minutes at most. We might schedule fine-tuning to avoid peak usage times to minimize any latency impact (unless we have a second GPU).

- **Parallelism and Scaling:** If the chatbot needs to serve many concurrent users, we would horizontally scale the inference service (multiple instances of the LLM container behind a load balancer). All instances would periodically fetch the latest adapter weights (for example, from a shared storage or model registry). When a new LoRA adapter is produced by the training process, it can be saved to a file or database; the inference servers detect the update (through a pub-sub or polling) and load the new weights. This can be done without restarting the service. This approach ensures all chatbot replicas remain in sync with the latest learned knowledge. The Postgres database can be scaled or at least tuned for the read-heavy load; caching layers (like embedding cache) can be used if needed.

Deployment on Limited GPUs:

Given a constraint of limited GPU resources (e.g., a single machine with 1-2 GPUs), we design the system to maximize usage of available compute: - **Quantization and batching:** We run the LLM in 4-bit mode (using libraries like bitsandbytes as noted). We also enable GPU memory optimizations such as gradient checkpointing during fine-tuning, and use the CPU offload if needed for optimizer states (bitsandbytes allows offloading 32-bit optimizer states to CPU RAM in QLoRA). This way, even a 13B model can train on a single 24GB GPU [42] [21]. Inference can also batch requests if there are bursts of queries, though for a chatbot latency is more critical so we likely handle them one by one or with small batches. - **Scheduling training vs inference:** If only one GPU is present, we schedule the fine-tuning jobs at low-traffic periods. For example, at midnight, we might briefly put the model in "update mode" (or route traffic to a standby instance or degrade to answering only from knowledge base templates) for say 10 minutes while we fine-tune on the day's feedback. Because the fine-tuning tasks are relatively light, this downtime can be minimal. Alternatively, use a time-sliced approach: the system could perform a quick fine-tuning on the fly if the load is low (since fine-tuning on 100 examples might only take 2-3 minutes, one could even do it in between user queries if there's an idle gap – though that's complex to orchestrate). - **Memory management:** We ensure that the adapter training code unloads the model after training to free memory, then the inference service reloads the new weights. We might use the same process to do both (loading and unloading as needed), or separate processes with IPC. Using separate processes could leverage disk or shared memory to pass the updated weights. On limited hardware, process isolation might help (to avoid fragmentation). - **Continuous vs iterative deployment:** We probably will adopt an **iterative update cycle (e.g. daily)** instead of truly continuous per user. This is a pragmatic choice: it gives time to accumulate enough feedback and to validate it, and it avoids constant GPU context switching. It also allows bundling multiple changes which is more efficient to train (since training on one example at a time is not GPU-efficient; better to batch many feedback examples). The architecture supports manual trigger as well – say a developer can trigger an immediate training run if they see a critical mistake that needs correction ASAP.

Update Frequency and Latency Considerations:

- **Feedback processing latency:** When a user submits feedback, the immediate actions (storing it, possibly adding a vector) are quick (milliseconds to a second). These won't noticeably affect the user. The user might even see an acknowledgement "Thank you for feedback" instantly.
- **Model update latency:** The time from feedback submission to that feedback influencing the model's answers is the more significant metric. We aim for a **short feedback-to-update cycle**, but not so short as to be unsafe. A reasonable goal is to deploy model updates **daily**. In a high-usage scenario with lots of feedback, we might even do it multiple times a day (say, every few hours). However, one must consider *evaluation and validation time* as well. Pushing updates too frequently might not allow

thorough testing. A daily cycle (perhaps nightly updates) is a good balance initially. As confidence in the automation grows, we could increase frequency.

- **Real-time adaptation possibility:** In some cases, for example if the knowledge base is updated (new document added) or a glaring factual error is discovered, the retrieval part updates immediately and the model might not even need fine-tuning (because RAG will pull in the correct info next time). So users can see improvements right away in terms of content. For model-based improvements (like not phrasing something rudely, or fixing a reasoning flaw), those appear after the next model update. If that's daily, a user might see the bot improve "the next day" on the same question they gave feedback on, which is quite a fast turnaround compared to traditional ML model update cycles.

- **Latency of responses:** We also monitor that the additional mechanisms do not slow down the chatbot's response time significantly. Using LangChain and RAG does have overhead: the vector search (tens of milliseconds typically) and the LLM generation. LLaMA-2 13B can generate a few tokens per second on a single GPU (depending on hardware). Usually, responses might take 2-5 seconds which is acceptable for a thoughtful answer. We can enable token streaming (LangChain supports streaming the LLM output tokens to the frontend) so that the user sees the answer being typed out. This improves the perception of speed. None of our feedback collection additions should add to the per-query latency (they run after the answer is delivered). So the user experience remains: ask a question, get an answer in a few seconds, then optionally provide feedback.

- **Monitoring and rollback:** We implement monitoring to catch any performance regressions after updates. If an updated model is slower or using more memory (which might happen if we significantly increase LoRA size or context length), we'll detect that. Because we maintain the capability to switch back to a previous model version easily (just re-loading older adapter weights), if something goes wrong (either a bug or a negative performance impact), we can rollback swiftly. This ensures high availability and reliability of the chatbot service.

In terms of concrete tools, we may use **Docker Compose or Kubernetes** to manage these components, especially if scaling. A Kubernetes CronJob could handle nightly training. We'd use Git or an artifact store to version-control the LoRA weights (so we have history of model versions). We also leverage existing **MLOps** practices: for example, evaluating the model on a validation set of Q&A pairs after each fine-tune to ensure it's improving the targeted issues without degrading others (this validation set could include some examples from earlier feedback as well as some standard queries).

Finally, security and privacy are part of the implementation strategy: ensure that the Postgres (with PGVector) is secure and encrypted, since it might contain proprietary documents and user chat logs. Access control is set such that only the application server can query the vector store (preventing data leakage). The user feedback, if sensitive, might be encrypted at rest. Compliance with data retention policies is also considered (maybe we don't keep raw conversations indefinitely, only the learned improvements).

By following this implementation plan, a technical team can **build and deploy the system on-premises or on a secure server with just a handful of GPUs**, leveraging open-source tools and libraries. The result will be a scalable RAG chatbot that **learns iteratively from its users**, offering improved answers and user experience over time, and doing so in a resource-efficient and controlled manner.

Conclusion

In designing this LLaMA-based RAG chatbot with real-time feedback adaptation, we created an architecture that tightly interweaves a **feedback loop** with the model's reasoning loop. The solution continuously collects user feedback (explicit ratings and implicit signals), rigorously filters and validates it, converts it into high-quality training signals, and then efficiently fine-tunes the model via LoRA/adapters – all while updating the retrieval knowledge base to reflect the latest information. This enables the chatbot to **get better with each interaction**, addressing its mistakes and aligning closer with user needs and values [27] [43]. Importantly, it achieves this adaptivity without requiring massive compute resources or prolonged offline training; instead, it uses **incremental, low-overhead updates** that fit into a local GPU setup [21].

The integration with LangChain ensures that the system remains modular and maintainable, benefiting from a robust framework for chaining LLM calls and managing data sources. By incorporating ethical safeguards at every step (from feedback filtering to cautious update deployment), the architecture avoids the pitfalls of naive online learning and focuses on **learning the right lessons** from users. The dynamic vector index keeps the model's knowledge fresh, complementing the fine-tuning which keeps the model's behavior refined.

Overall, this architecture empowers technical teams to build a **scalable, self-improving RAG chatbot** – one that can be deployed in environments with limited GPUs (or strict data control requirements), yet still progressively enhance its performance through real-time user feedback. It demonstrates how modern techniques like RAG, PEFT (LoRA), and continual learning can be combined into a virtuous cycle: as users engage and provide feedback, the system becomes more accurate, aligned, and responsive, thereby driving higher user satisfaction and trust in the AI assistant [44] [19].

By following the strategies outlined above, an applied research or engineering team can implement this system and adapt it to their specific domain (be it customer support, internal knowledge bases, etc.), resulting in a chatbot that not only serves answers but **evolves** to deliver *better* answers with each day of use – all while upholding the principles of efficiency, safety, and scalability.

Sources: The design draws on established approaches in RLHF and RAG literature, including the integration of human feedback to reduce hallucinations [45] and improve alignment, the use of LoRA for rapid fine-tuning on low-resource hardware [21], and practical insights from industry implementations of feedback-driven improvement [37] [13]. The above architecture is informed by these principles and tailored for a LLaMA-centric, locally deployable scenario.

---

1 2 3 14 15 16 17 23 26 28 30 37 45 Improve LLM performance with human and AI feedback on Amazon SageMaker for Amazon Engineering | AWS Machine Learning Blog
https://aws.amazon.com/blogs/machine-learning/improve-llm-performance-with-human-and-ai-feedback-on-amazon-sagemaker-for-amazon-engineering/

4 5 13 31 32 40 41 How to leverage user feedback into my LLM application? : r/LangChain
https://www.reddit.com/r/LangChain/comments/16n4fug/how_to_leverage_user_feedback_into_my_llm/

6 7 8 12 18 19 20 43 44 How to Track LLM User Feedback to Improve Your AI Applications - DEV Community
https://dev.to/lina_lam_9ee459f98b67e9d5/how-to-track-llm-user-feedback-to-improve-your-ai-applications-1a08

9 10 11 25 29 38 Why can't LLMs be continuously trained through user interactions? : r/OpenAI
https://www.reddit.com/r/OpenAI/comments/1gmf4ox/why_cant_llms_be_continuously_trained_through/

21 22 24 Memory requirements for fine-tuning Llama 2 | by Sri Ranganathan Palaniappan | Polo Club of Data Science | Georgia Tech | Medium
https://medium.com/polo-club-of-data-science/memory-requirements-for-fine-tuning-llama-2-80f366cba7f5

27 33 (PDF) Retrieval Augmented Generation-Based Chatbot for Prospective and Current University Students
https://www.researchgate.net/publication/392785685_Retrieval_Augmented_Generation-Based_Chatbot_for_Prospective_and_Current_University_Students

34 39 Build RAG using Llama 2, Vector Store, and LangChain | JFrog ML
https://www.qwak.com/post/rag-llama-langchain

35 Fine-tune your LLMs with LangSmith and Lilac
https://blog.langchain.com/fine-tune-your-llms-with-langsmith-and-lilac/

36 Self-Reflective RAG with LangGraph - LangChain Blog
https://blog.langchain.dev/agentic-rag-with-langgraph/

42 Fine-tuning Llama-3–8B-Instruct QLORA using low cost resources
https://medium.com/@avishekpaul31/fine-tuning-llama-3-8b-instruct-qlora-using-low-cost-resources-89075e0dfa04