ChatGPT

Open-Source Repositories and Tools for the System Architecture

Below we organize the top open-source code resources by each module of your architecture – from policy-as-code orchestration and guardrails to retrieval, emotion classifiers, letter rewriting, multi-agent flows, evaluation, privacy, and more. Each entry includes a brief annotation, license, and notes on integration ("how to plug in"). We focus on mature, permissively-licensed projects that can be integrated with minimal changes, aligning with your **strategy-first, state machine** approach.

Dialogue Orchestration (Policy-as-Code & State Graphs)

- **LangChain LangGraph** (MIT License) – A low-level orchestration framework for **building stateful conversation agents as graphs** [1]. LangGraph supports **finite-state workflows**, multi-turn tools, and branchable dialog flows on top of LangChain. You can define conversation stages/nodes (e.g. mood check, agenda setting, etc.) and transitions between them using Python. It integrates with LangChain memory and LangSmith for debugging [2] [3]. *How to integrate:* Install via `pip install langgraph` [4] and define your therapy session stages as a graph. For example, model your 8 CBT stages as LangGraph "agents" with structured prompts, then use LangGraph to manage stage transitions (ensuring the AI moves through MoodCheck → Agenda → … → Feedback in order). LangGraph's durability and human-in-the-loop features help maintain long sessions [5] [6].

- **Rasa Open Source** (Apache 2.0) – A mature conversational AI framework with **state machine dialogue management**. Rasa uses intents, entities, and **story graphs** to enforce conversation structure. While heavy-weight, it excels at predictable control (forms, slot filling) and can incorporate LLMs via a **hybrid** approach [7]. *Integration:* Use Rasa if you need a robust state machine for dialogues – you can design your therapy dialog as stories or rules, and even call an LLM (via a custom action) for free-text responses. Rasa ensures important steps (e.g. consent, safety check) aren't skipped. It's production-proven, though it may require more initial setup.

- **RasaGPT** (MIT License) – A **boilerplate combining Rasa and LangChain** [8]. It uses Rasa's dialogue policies plus LLM-based retrieval (via LlamaIndex) and a FastAPI backend. RasaGPT essentially merges classical dialogue state management with LLM flexibility. *Integration:* This can serve as a template – it already includes Rasa for structuring flows (intents like greet/affirm/ask_question) and uses LangChain for knowledge base queries [8]. You could adapt RasaGPT's structure to your domain (therapy/parenting) by training Rasa NLU on relevant intents (e.g. user asks for coping strategy, user shares feeling) and plugging in your vector DB for retrieval. It provides a headless (no UI) but modular starting point.

- **pyTransitions** (MIT) – A lightweight Python **state machine library** ideal for coding custom dialogue graphs. It lets you define states and transitions with conditions. *Integration:* Use pyTransitions to implement a simple deterministic dialog flow (e.g. a **"scripted" crisis flow** that must ask: "Are you feeling safe?" and branch accordingly). You can invoke LLM calls inside state actions. This is lower-level (no ML/NLU built-in), but gives full control for critical flows (like a

DIFF_DIAG_GATE node ensuring crisis protocol). It's useful for smaller stateful sequences inside your larger LangChain framework.

- **XState (JavaScript)** or **AI Flow Markup** – If you ever consider a visual or JSON/YAML representation of the conversation graph, tools like XState (statechart library) or writing a custom YAML schema (`graph.yaml`) could help. For instance, you could design a YAML that lists states (e.g. `LETTER_COACH`, `SCHOOL_MEETING`) and valid next states, then interpret it in Python to enforce allowed transitions. This "policy-as-code" YAML can be source-controlled and reviewed by domain experts. *(No ready-made library in Python for YAML-driven LLM flows exists yet, but you could implement one using pyTransitions or LangGraph's API.)*

Why these? LangGraph stands out for natively handling **LLM agent orchestration** (with support from LangChain team) [9] [10]. Rasa brings **battle-tested dialogue management** plus a growing focus on LLM integration [11]. RasaGPT exemplifies bridging rule-based and generative approaches. These ensure your "strategy-first" logic (e.g. structured CBT session flow) is enforced around the LLM.

Guardrails and Safety Enforcement (Policy DSL, Moderation)

- **NVIDIA NeMo Guardrails** (Apache 2.0 License) – A toolkit to **add programmable guardrails** (a.k.a. "rails") to LLM-based chatbots [12]. It introduces a policy DSL called **Colang** for defining allowed/prohibited bot behaviors and dialog flows. For example, you can write rules like *"If user mentions self-harm -> respond with empathy + show crisis resource"* or *"Avoid discussing politics"*. Guardrails sits between your app and the LLM to intercept inputs/outputs and apply these rules [13] [14]. *Integration:* Write Colang files (`*.rail`) specifying your safety policies (e.g. a file for **crisis intervention** that triggers an on-call flow, a file for **jurisdictional legal advice blocks**). Load these with NeMo's Python API to wrap your LLM calls [15] [16]. Minimal code changes are needed – you replace `openai.Completion.create(...)` with a guardrails `LLMRails.generate(...)` which ensures outputs obey your rules [17]. NeMo Guardrails also provides **jailbreak/prompt-injection protection** by filtering or altering prompts known to cause vulnerabilities [18], and can enforce structured dialogs. It's in beta, but powerful. (In production, use carefully: NeMo's docs note it's under active development [19].)

- **Guardrails AI (Shreya's library)** (Apache-2.0) – Often just called "**guardrails**", this Python library validates and constrains LLM outputs using **structured output specifications** (like JSON schemas or custom check functions). You can define an output schema (e.g. a JSON with fields `{"advice": string, "tone": enum["NVC","harsh"]}`) and Guardrails will force the LLM to comply, retrying or fixing outputs that don't validate [20] [21]. It also supports plugging in validators for content (e.g. a PII detector or toxic phrase detector). *Integration:* Use Guardrails to ensure the assistant's responses meet a format and style. For example, for letter rewriting, you might require output JSON: `{ "rewritten_letter": "...", "strategy": "BIFF" }` and add a custom validator that checks politeness or absence of forbidden words. If the LLM's output fails, Guardrails can **"fix"** it automatically or prompt for correction [22] [23]. Guardrails comes with a **PII removal plugin** (see below) and other validators (e.g. length, JSON format). It's easy to integrate: you wrap your LLM call with a Guardrails `Guard` that enforces a provided schema or script.

- **Open Policy Agent (OPA)** (Apache-2.0) – A general-purpose policy-as-code engine (with its own declarative language, Rego) often used for enforcing rules in infrastructure. Here, OPA can be repurposed to enforce **non-ML business rules** around your chatbot. For example, you could write OPA policies for **jurisdiction-specific advice**: if user location is EU and topic is legal, then

refuse answer. Or a policy that *"If user is under 18, do not discuss certain topics"*. *Integration:* OPA runs as a lightweight engine (or service) where you load rules and query decisions. You might have your app send conversation metadata to OPA before responding. If OPA returns "deny", the bot can adjust or refuse. While not LLM-specific, OPA gives a robust rule layer separate from the model – useful for compliance (e.g. parental consent rules). It can be part of your pipeline: **LLM -> OPA check -> user**. (The overhead is minimal, as OPA rules are fast logical checks.)

- **Content Moderation Models** – Open-source classifiers for toxicity and sensitive content can act as guardrails. For example, OpenAI's moderation (if using their API) or Meta's **Llama-2-Chat built-in guardrails**, or the **LlamaGuard** model by Meta (a 7B fine-tuned model for moderation) have been released. There's also **Perspective API** (Google's toxicity scoring) – not open source but free for moderate use. If offline is needed: see **Detoxify** under the *Emotions* section for toxicity filtering of user inputs **before** they hit the LLM (to avoid certain prompt attacks or just to log flagged content).

- **NVIDIA Garak** (Apache-2.0 License) – An **LLM "vulnerability scanner"** for red-teaming your model [24]. Garak provides a library and CLI to run a suite of known prompt attacks (jailbreaks like "DAN", logic puzzles, etc.) against your chatbot to see where it fails. It's not integrated at runtime (it's a testing tool), but it's invaluable for **hardening**. *Integration:* Use Garak during development/CI to probe your system for safety holes. For instance, add a CI step where Garak runs a set of prompt injection attempts on your deployed model (or on a local instance). If Garak finds a jailbreak that gets the bot to violate rules, you can then adjust your prompts or guardrails. Garak comes with scenarios including prompt leaks, refusal-bypass, toxicity induction, etc., and it reports vulnerabilities discovered.

- **Rebuff (Protect AI)** (Apache-2.0, *archived*) – A **prompt injection detection framework** [25] [26]. It used a multi-layer approach: heuristic filters (e.g. look for strings like "ignore previous instructions"), an LLM-based classifier that detects likely injections, a vector database of known attack signatures, and **canary tokens** (injecting hidden words to detect instruction leaks) [26]. *Integration:* Although the project is now archived (no longer actively maintained) [27], its ideas can be replicated. For example, you could implement a simple *heuristic filter* on user input (if input contains "ignore all previous" or unusual Unicode control chars, flag it). You could maintain an *"attack vector DB"* – e.g. store embeddings of known bad prompts and check new inputs via similarity (if very close, likely an injection). And the *canary token* idea: append a known "secret" sentence in system prompt and check if the model ever outputs it – indicating a context leak. Rebuff's code can be used as reference [28] [29]. It's a bit advanced, but if prompt injection is a major risk (given you allow tool use or internet access), these techniques add defense in depth.

Why these? Safety is paramount in mental health applications. NeMo Guardrails and Guardrails AI **let you declaratively specify policies** (one at design-time via Colang, the other at run-time via schema/validators). They directly address the need for controlling LLM output on sensitive topics (self-harm, legal, etc.) [30] [14]. The OPA addition underscores that not all rules need to be learned by the AI – some can be **hard-coded policies** (e.g. age or jurisdiction checks). Meanwhile, Garak and Rebuff (though the latter is prototypical) cover the adversarial angle, ensuring your guardrails hold up under intentional misuse. Combining these, you get *multi-layered safety*: **prevent** bad outputs with guardrails, **detect and respond** to attacks or unsafe inputs with classifiers, and **continuously test** with red-team tools.

Retrieval-Augmented Generation (RAG) & Knowledge Integration

- **Haystack by deepset** (Apache-2.0 License) – A popular framework for **building RAG pipelines**: it supports document ingestion, embedding-based retrieval (via ES, FAISS, Weaviate, etc.), and generative QA on top [31]. It's production-grade and domain-agnostic. Haystack provides pipelining of components: e.g. a Retriever (BM25 or DPR) + Reranker (cross-encoder) + Reader (LLM or smaller QA model). *Integration:* Use Haystack to give your bot a long-term memory or knowledge base. For instance, load a **therapy corpus** (e.g. psychology articles or your own guidelines) into Haystack's DocumentStore. At runtime, when the user asks a factual question ("What's the DSM-5 criteria for X?"), query Haystack: it will retrieve relevant text and you can either feed it into the LLM prompt (retrieval-augmented generation) or use Haystack's built-in `GenerativeQA` node to do it for you. Haystack has **pipelines for chat** too, maintaining history. It also integrates with **huggingface models** for embedding or reader, so you could use local models for privacy. Being an end-to-end toolkit, it can handle evaluation of the QA as well. deepset has a vibrant community and even a **Haystack Hub** for healthcare.

- **LlamaIndex (GPT Index)** (License: Apache-2.0 for open-source version) – A toolkit to **connect LLMs with external data** (documents, DBs) in a simple way. It offers high-level abstractions: **indices** like VectorStoreIndex, ListIndex, etc., and can automatically create query engines. *Integration:* LlamaIndex can wrap around your text data sources (e.g. a collection of therapy session transcripts, or parenting tip articles) and provide a natural language query interface. If, say, the user's query falls under "factual info" category, you can route it to a LlamaIndex QA retrieval instead of the normal chat pipeline. It's very flexible – you can compose indices (e.g. an index per topic and a meta-index to choose among them). It has caching and supports many vector stores (FAISS, Pinecone, Weaviate). Note: ensure the license terms fit your use (the open core is Apache-2, but advanced features may require a pro license).

- **Weaviate / Qdrant / Milvus** – These are open-source **vector databases** (Weaviate (Apache-2), Qdrant (Apache-2), Milvus (Apache-2)) you can use to store embeddings for semantic search. Rather than a simple in-memory FAISS, a vector DB offers persistence, filtering (e.g. "only docs tagged as `parenting`"), scaling, and hybrid search. *Integration:* If your project will have a lot of content (or you want to quickly **update knowledge by adding docs**), consider running a vector DB. For example, if the bot should cite **local legal statutes or school policies**, you can index those text snippets in Qdrant with metadata (region='LV', type='school_policy') and on queries, do a similarity search. All aforementioned RAG frameworks (Haystack, LlamaIndex, LangChain) support these backends easily. This external knowledge greatly enhances accuracy and trustworthiness in advice (reducing hallucinations by grounding answers [32]).

- **RAG Evaluation – RAGAS** (MIT License) – *Ragas* is an evaluation toolkit specifically for RAG pipelines [33]. It can compute metrics for the retrieval component (e.g. recall, MRR using labeled data) and for generation (e.g. correctness via LLM-based scoring). It also provides a CLI to generate test sets. *Integration:* As you build RAG into your bot, use RAGAS to evaluate if the system is actually fetching relevant info and answering accurately. For example, after adding a knowledge base of mental health FAQs, you can run RAGAS with a set of known Q&A pairs to see how often the correct passage is retrieved and the answer is correct [34]. RAGAS can integrate with LangChain and LangSmith for logging results [35]. It's a **guardrail for truthfulness**, helping you tune embeddings, prompt, or decide on a better retriever.

- **Retrieval Enhancements**: you might incorporate a **Reranker model** (e.g. a cross-encoder like `ms-marco-MiniLM-L6` from HuggingFace) to re-rank retrieved passages by relevancy –

Haystack supports this out of the box. Also consider **chunk splitting** strategies: using *semantic splitting* (as noted) to chunk text by meaning rather than fixed length. This prevents the bot from missing context due to awkward splits. LangChain has `RecursiveCharacterTextSplitter` and there are community notebooks for semantic text splitting [36]. Ensuring each chunk is self-contained (e.g. not cutting a sentence in half) improves retrieval quality.

Why these? They provide **production-ready infrastructure for knowledge retrieval**. Haystack in particular is proven in many QA apps (with plugins for document stores, including an option for **embedding-based similarity + keyword fallback**). LlamaIndex offers a faster ramp-up if you prefer Pythonic integration and incremental adoption. The vector DBs ensure your memory can scale and be updated (useful for dynamic content like new articles or user-specific info). Given mental health assistants must often provide *accurate information* (e.g. about coping techniques or crisis steps), RAG can ground the LLM's answers in vetted sources [37]. It also supports **citations**: you can have the bot output references to the retrieved texts (and even show quotes) – increasing user trust.

Emotion, Tone & Crisis Detection (Classifiers & Models)

- **Detoxify (UnitaryAI)** (MIT License) – A set of **pre-trained models for toxicity detection** [38]. Detoxify models (like `unbiased-toxic-roberta`) predict probabilities for toxic content categories (insult, threat, hate, etc.). *Integration:* Use Detoxify to **screen user inputs and bot outputs for toxicity**. For example, if a user is extremely angry or using abusive language, you might route to a special de-escalation response. Or ensure the bot's outputs maintain a neutral/non-toxic tone. It's a Python pip package – just load the model (PyTorch) and call `predict([text])`. This can serve as a **pre-check** (before LLM): if user's message is flagged e.g. 95% toxic, maybe the bot should address the anger first or enforce a safety guideline. Similarly, as a **post-check** on LLM output, prevent any policy-violating content from being shown. Detoxify was trained on Jigsaw's large toxic comment dataset, so it's robust.

- **GoEmotions (Google)** – A dataset of 58k Reddit comments labeled with **27 emotion categories** (plus neutral) [39], and corresponding models. There are open-source classifiers fine-tuned on GoEmotions (e.g. a BERT or RoBERTa on HuggingFace). *Integration:* Incorporate an **emotion classifier** to detect the user's emotional state each turn: e.g. user message -> classify as {sadness: 0.8, anger: 0.1, etc}. This can feed into your strategy (the *state machine* could transition based on emotion intensity – for instance, a high sadness score might trigger the **"empathy subroutine"** in your graph). You can also label the assistant's own responses or target tone (ensuring it matches or appropriately responds to the user emotion). Many off-the-shelf models exist (check HuggingFace Hub for "goemotions"). Since these models are lightweight (BERT base), you can run them alongside the LLM.

- **Suicide Risk / Crisis Classifiers** – This is critical: detect **suicidal ideation or self-harm expressions** in user messages. Datasets like the *Suicide Ideation Dataset* from Reddit (e.g. 200k posts balanced between ideation and not) have yielded fine-tuned BERT models [40]. For example, *"Suicidal-BERT"* (on HuggingFace) predicts if a text is suicidal vs not with ~97% accuracy [41]. *Integration:* Use such a model as a **DIFF_DIAG_GATE** – if the user message is classified as high risk, you immediately route to a crisis handling flow (bypass normal dialog). The classifier can run before the LLM each turn. If triggered, you might use a special prompt or template designed for crisis (e.g. encourage seeking help, ensure not to give harmful advice). This gate can also escalate (maybe alert a human moderator if available, or display emergency resources). Given the sensitivity, having an explicit model for this is safer than relying on the LLM to infer risk from prompt instructions. You might also incorporate **self-harm content detection** (like

classifying content into categories: none, ideation, explicit plans, etc., which some research models do).

- **Empathy and Emotion Analysis** – Beyond raw emotion labels, you might want to measure how *empathic* the assistant's responses are, or guide it to be more empathic. Check research like Sharma et al. (2020) on **empathetic dialogue**; they used classifiers for *empathy, contextual understanding, and exploration* in conversations. For instance, an **empathy classifier** (RoBERTa model) can score a reply on how empathic it is [42]. Intel's **PoliteGuide/Politeness** resources (Stanford Politeness Corpus) provide models to detect politeness strategies. *Integration:* Use these as metrics or feedback signals. E.g., after the LLM generates a response, run an empathy classifier – if it scores low and user is in distress, you could have a second pass where the LLM is instructed to *"try again with more empathy"*. This effectively implements a **reflection loop** where the bot evaluates its own tone. Similarly, a **politeness/toxic classifier** on the bot's output ensures it never sounds harsh or rude (should be mostly unnecessary if your prompts are good, but a nice safety net).

- **Tonal Rewriting Tools** – There are also style transfer models (e.g. for formality or politeness). For example, a GPT-J-6B fine-tuned to generate *polite paraphrases* [43]. These are less common, but if you find one (or fine-tune a smaller model yourself on parallel data), you could have a pipeline: raw LLM output -> style transfer model -> final output, to guarantee a certain tone (like always BIFF-friendly or clinically neutral). This is an advanced optional step; often simpler to prompt the main LLM for the desired tone directly.

Why these? Emotions are the currency of therapy dialogues. Recognizing the user's emotional state lets the system **adapt responses dynamically** (just like a human therapist would respond differently if a client sounds depressed vs angry). The **GoEmotions** taxonomy is broad enough to capture nuanced feelings beyond basic sentiment. The crisis classifier is a must for user safety – it's a proven approach in recent research to mitigate harm (many mental health chatbot papers include an explicit suicide risk detector). Toxicity and politeness detection ensure the assistant **maintains a respectful, calming tone**, even if the user doesn't. Overall, these classifiers act as an **"emotional intelligence module"** for your AI: monitoring and guiding the affective quality of the conversation in real-time.

Trauma-Informed and Crisis-Sensitive Behavior

Designing the bot to be **trauma-informed** means the language and flow should avoid triggering or re-traumatizing users. While not a single software package, there are guidelines and you can enforce them through prompts/policies:

- **Trauma-Informed Language Guidelines** – For example, a draft standard for mental health chatbots suggests: *"Use language/tone that avoids triggering or retraumatizing users, making sessions feel emotionally safe"* [44]. Concretely, this means the bot should **not press for details of trauma**, should frequently offer the user control ("We can stop if you feel uncomfortable"), and use validating, non-judgmental phrases. *Implementation:* Incorporate these principles in your system prompts and guardrails. You could create a **style guide** that the LLM must follow (e.g. *"The assistant should never say phrases that minimize the user's experience or blame them. The assistant should emphasize the user's safety and choice."*). Use Guardrails/Colang to ban or flag certain phrasings. Also, train the bot (via few-shot examples) on trauma-sensitive responses (for instance, example dialogues where a user discloses abuse and the assistant responds gently with validation and optional next steps, no explicit details unless user wants).

- **Crisis Flow & Escalation** – If the user indicates they are in a crisis (suicidal ideation, panic attack, etc.), the bot should follow a **scripted de-escalation flow** (this ties in with the crisis classifier above). This flow can be a simple finite state machine: Check for imminent danger → Encourage them to seek help → Provide emergency resources → Ask if they have someone they can call, etc., while **staying with them** in a supportive manner. You can code this as a separate function or use NeMo Guardrails to define a special response pattern on such triggers. Ideally, have a list of crisis resources (perhaps region-specific) that the bot can provide. Make sure the **rails.colang** or policy rules require this behavior on self-harm content. Also ensure the bot **never** gives harmful instructions (this is obvious but must be guarded – e.g. a user with trauma might ask for unhealthy coping like self-medication; guardrails should forbid suggesting anything dangerous).

- **Avoid Re-Traumatization** – The bot should not force the user to relive trauma. Technically, you can handle this by *not prying into traumatic details unless the user volunteers them*. If the user does start describing traumatic events, the bot should respond with empathy and perhaps gently steer toward feelings or coping, rather than asking for more graphic detail. You might incorporate a **content filter** that if a message contains very graphic or violent content, the bot responds with extra care or maybe shorter responses (the idea being to not inadvertently dwell on graphic details). OpenAI's content policy style can be mimicked here: for certain categories like sexual abuse, perhaps the bot provides a brief empathic acknowledgement and offers professional help rather than discussion (to avoid unintentional harm).

- **Training/Examples** – If possible, include example dialogues in your prompt or fine-tuning data that demonstrate trauma-informed techniques: use of grounding exercises, giving the user choices ("Would you like to try a coping exercise now?"), and **no judgment**. This will guide the model's generation.

In summary, while there isn't a plug-and-play "trauma mode" library, you implement it via **rules, style guidelines, and curated prompt data**. Use the policy frameworks above to hard-code critical aspects (like always include a disclaimer and referral in sexual trauma cases, etc.). Keep testing with diverse users (including those with trauma histories if possible, or subject matter experts) to see if any responses could be improved to meet trauma-informed standards. The outcome should be a bot that *feels safe* – as one guideline put it, "emotionally safe for all backgrounds" [44] – and this can be achieved by carefully shaping its language and choices in code.

Therapeutic Letter Rewriting (NVC, BIFF & Coaching Communications)

One of your unique features is helping users rewrite messages (to ex-spouse, school, etc.) in healthier tones (NVC, BIFF). Here are resources and approaches to power that:

- **Nonviolent Communication (NVC) Prompts** – You can implement NVC style rewriting by prompting the LLM with a structured format: identify observation, feeling, need, request. You might not find an off-the-shelf code library that does NVC transformation, but you can create a prompt template or small function. *Integration:* For example, take the user's angry paragraph, and feed the model a system prompt: *"You are an assistant skilled in Nonviolent Communication. Rewrite the user's message following NVC: 1) State observations without judgment, 2) Express feelings, 3) Express needs, 4) Make a clear request."* Provide an example in the prompt if possible. The model will then output an NVC-style "I-statement" version. This is purely prompt engineering; GPT-4 does quite well on this if guided. You can also detect certain patterns (like accusatory "you"

statements) and have rules to remove them. The classifier tools above (like politeness or sentiment) could double-check that the tone improved.

- **BIFF Response Templates** – BIFF (Brief, Informative, Friendly, Firm) is from the High Conflict Institute. While no direct code, you can incorporate BIFF principles similarly. *Integration:* Create an LLM prompt that instructs: *"Rewrite the following text to be BRIEF, just the facts (Informative), with a friendly tone, and firm (final) in conclusion. Remove any blame or negative language."* The assistant can produce a BIFF compliant reply. You might maintain a few **exemplar pairs** (original vs BIFF rework) and use them as few-shot examples in the prompt to steer the model. Another strategy: break it into steps – the model first summarizes the key point (Brief/Informative), then separately rephrases with positive tone (Friendly) and adds a firm closing. A simpler heuristic approach could complement the AI: e.g. use a **prose linter** to catch unnecessary sentences or aggressive adjectives.

- **Proselint / LanguageTool / alexjs** – These are **linters for text style** that can flag issues like overly complex sentences, passive voice, or even emotionally charged language. **Proselint** (MIT License) is a Python linter for English prose that detects various writing issues [45] (e.g. biased language, jargon, needless intensifiers). **LanguageTool** is multilingual and can check for politeness (to some extent) and formality, though it's more grammar-focused. **alex.js** is a JavaScript tool that highlights insensitive language (like gendered terms, etc.). *Integration:* After the LLM produces a rewritten letter, run it through Proselint to catch things like condescending tones or cliches. Proselint might say "avoid the word 'obviously'" which indicates a judgmental tone – you can then prompt the LLM to remove or replace that. Essentially, use linters to **critique and iterate on the output**. This can be automated: if Proselint returns warnings above a threshold, have the LLM "revise to address these warnings". (In a user-facing UI, you could even show suggestions to the user, but for automation, it's easier to loop back into the model.)

- **Toxicity and Politeness Check** – Tieing in earlier tools: after rewriting, ensure the new letter scores low on toxicity and high on politeness. For example, Stanford's **Politeness classifier** (available via ConvoKit library) could rate the letter. If it's still somewhat impolite (maybe the model left a subtle dig), you could instruct another revision. This is like an automated quality filter before showing the result to the user.

- **Formal/Professional Style Transfer** – Some NLP research deals with style transfer, e.g. transforming informal to formal. There might be pre-trained models on the **GYAFC dataset** ("Get Your **@#$** Together Charlie" from Yahoo answers) that convert rude to polite or informal to formal. One such model is an older one by IBM or University of Toronto (some are on HuggingFace). If you find a stable one, you could feed the user's text into it for a first pass at "neutralizing" tone, then refine with NVC. But in practice, GPT-3.5/4 can do this in one go with proper prompting, so an extra model isn't strictly required.

- **Coaching Hints** – Instead of fully automatic rewriting, you might sometimes present the user with suggestions: e.g. highlight sentences that are judgmental and suggest "Consider rephrasing this as an I-statement about your feelings." This is a semi-manual approach and could be implemented via an LLM (by asking it to list problematic phrases and fixes) or via simpler pattern matching (if you have a knowledge base of common hostile phrases vs their neutral counterparts). For example, if the text says "You never help with the kids!", a simple rule or regex could detect "you never" and flag it. The LLM or your code can suggest "Try starting with 'I feel overwhelmed when I don't get help with the kids.'"

Why these? Many high-conflict communication experts (like Bill Eddy of HCI) emphasize rewriting as a structured process. By leveraging LLMs with clear instructions, you can automate a lot of it. The combination of **LLM + linting** is powerful: the LLM does the heavy linguistic work, and the linters/classifiers provide guardrails to ensure the output meets specific criteria (briefness, positivity, etc.). This approach keeps the rewritten letters **production-ready** – essentially passing automated "quality checks" before the user sees them.

Also, all tools here are either code libraries (Proselint, LanguageTool) or straightforward prompt patterns, which aligns with quick integration. Proselint is easy to call in Python and extend with custom rules (you could even add a rule: "if sentence starts with 'You,' flag it, since in BIFF we prefer starting with 'I' or neutral subject"). These rewrites and checks ensure that by the time the user gets the suggestion, it's polished and aligned with best practices in conflict communication.

Parenting Support & Coaching Modules (CBT, MI, Goal-Tracking)

This aspect involves interactive exercises and coaching, often structured (like CBT worksheets, MI techniques, goal setting frameworks):

- **CBT Thought Records & Exercises** – A thought record is a step-by-step guide for cognitive restructuring. There's an open-source project **CBTree** (MIT License) that implements a web app for creating and managing CBT thought records [46]. It basically asks the user a series of questions (Situation -> Thoughts -> Emotions -> Challenge -> Outcome) with a defined route sequence [47]. *Integration:* You can take inspiration or even code from CBTree: it has a backend in Python and a frontend in TS. For your chatbot, you don't necessarily need a UI – you can convert the flow into a conversational form. Use a **state machine or a LangChain "chain"** to guide the user through the steps. For example, when a user is distressed, the bot could say "Let's try a quick exercise. Can you describe what situation is bothering you in one sentence?" (state 1), then "What thought is going through your mind?" (state 2), etc. The state logic ensures each question is asked in order and uses the user's previous answer in the next prompt if needed. This could be implemented with LangChain's multi-turn prompt templates or just manual logic. The key is you have a **template for each step** and the bot patiently waits for the answer, possibly summarizes it, and moves on. Having these exercises coded out (maybe in YAML or Python dict) would make it easy to invoke them as needed (like a subroutine in your dialogue graph).

- **Motivational Interviewing (MI) Strategies** – MI uses techniques like open questions, affirmations, reflective listening, and summaries (OARS). To integrate MI, you might not find a code library, but you can define **prompting strategies**. For instance, an MI reflection can be prompted by taking the user's last message and asking the LLM: *"Respond with a reflective listening statement that paraphrases the user's content and emotion, without judgment or advice."* You can create a function `generate_reflection(user_message)` that uses the LLM with such a prompt. Also, **intent recognition** can help: if a user message seems to contain *sustain talk* ("I don't think I can change"), the bot can respond with *change talk elicitation* (like "What are some reasons you'd want to make this change?"). If you want to be systematic, you could train or use a classifier to detect change talk vs sustain talk, but simpler: use keywords or the LLM itself (ask it: "Classify this as sustain-talk or change-talk"). *Integration:* Weave MI micro-skills into the conversation by occasionally replacing a normal response with a reflective one, especially when the user expresses ambivalence. The **LangChain Agent** could have a tool like `reflect(message)` that the agent chooses when appropriate – or you handle it in your graph logic (e.g., a node that checks `if last_user_message.contains("can't" or "don't want") then do MI_reflection`).

- **Goal Setting and Tracking** – Supporting SMART goals or habits might require the bot to help user define a goal and then check in later. This is partly a UI/storage challenge (you need to save the goals somewhere). For open-source code: you might look at habit tracking apps or "coach bots". Notably, there's **open-source habit trackers** (mostly mobile apps though) and some research on goal-setting chatbots. *Integration:* Use a simple database or even a JSON file to store user's goals (with user consent). You can create an interface: "Goal Coach" node where the bot asks the user to formulate a goal (Specific, Measurable, etc. each attribute could be one question). That becomes a structured record. Then you could set up a mechanism (maybe a scheduled job or just next session detection) to bring it up: e.g. if user returns after a week, the bot greets: "Last time you set a goal to walk 3 times a week. How has it been going?" This scheduling can be handled outside the bot (cron job triggering a message) or simply triggered when the user next speaks (the bot can check last goal timestamp). There's not a particular repo that does this out of the box, but the logic is straightforward to implement with standard Python scheduling or using an existing task scheduler like APScheduler.

- **Parenting Coach Scenarios** – Some specific parenting situations (e.g. handling a tantrum, co-parenting conflict) could be handled by scenario scripts. I didn't find open libraries specifically for "parent coaching dialogs," but resources like Triple P or CDC parenting tips exist as text. You might encode a few as **retrievable Q&A**: for example, user says "My teenager won't listen" – the bot can retrieve an FAQ answer on that scenario and then discuss it with the user. If you have access to any public domain parenting guides, consider adding them to your RAG knowledge base. Another angle: incorporate **NVC for parenting** – if the user drafts a message to a child's other parent, combine the letter rewriting approach with parenting tone (some sources available in Russian as well, e.g. "parallel parenting agreements" templates which you mentioned – those might be PDFs you can glean language from).

- **Exercises and Journaling** – The bot could offer guided journaling or mindfulness exercises (e.g. "Let's do a 5-minute breathing exercise"). There are open resources like **Mindfulness scripts** (some on GitHub or free sites). Implementation is similar to CBT exercises: a predefined script that the bot follows, possibly adapting to user input. E.g., a journaling exercise might simply prompt the user to write for a few minutes about something positive, then the bot offers to highlight strengths in what they wrote (LLM can do sentiment analysis or strength spotting). For multi-turn exercises, controlling the flow is important – use your orchestration framework so the model doesn't wander off script.

Why these? These interventions (CBT, MI, etc.) are evidence-based and giving your bot the ability to conduct them makes it far more effective than generic chat. By leveraging open examples like CBTree, you avoid reinventing the flow logic for something like a thought record – you can adapt that code to a conversational format [47]. MI techniques increase user engagement and are crucial for motivational contexts like habit change; implementing them via prompt patterns is relatively low-effort with high payoff. The key is these are **structured interactions** that can be pre-designed and then just triggered within the chat when appropriate (almost like mini-apps inside the chat). Open-source code gives you templates (like CBTree's questions or known OARS examples) to ensure fidelity to the therapeutic technique.

Just-In-Time Adaptive Interventions (JITAI) & Personalization

JITAI refers to delivering the right intervention at the right moment, often based on context data, and often using **bandit algorithms or triggers**:

- **Contextual Bandit Libraries (MABWiser)** – To decide *when* and *what kind* of intervention to deliver, you can use multi-armed bandit algorithms (which learn optimal actions through feedback). **MABWiser** (Fidelity Labs, Apache-2.0) is a Python library for contextual multi-armed bandits, supporting strategies like epsilon-greedy, UCB, Thompson sampling with or without context features [48] [49]. *Integration:* For example, define arms = {"send motivational message", "stay silent"} for a daily check-in, context = user's recent mood or engagement. The bandit can learn if sending a prompt at evening results in better user response vs morning. Or arms = {suggest CBT exercise, suggest mindfulness, just empathize} with context = user's current stress level. Over time, it personalizes which intervention works best for that user. You'd need to define a reward signal (maybe user's rating or whether they respond the next day). MABWiser's API lets you update the model with new observations easily. This adds a **learning loop** to your system beyond the LLM.

- **Rule-based Just-In-Time Triggers** – In absence of complex sensors (like a wearable providing stress levels), you might trigger interventions based on conversation cues or user's routine. For instance: *"If user hasn't checked in for >3 days, send a nudge"* or *"If user mentions relapse or extreme language, immediately offer an exercise."* These can be coded with simple scheduling and regex on messages. Python's **APScheduler** can schedule jobs (send message X every Monday 9am, etc.). If you are sending proactive messages (ensure user consent), you could integrate a messaging scheduler that pings the user with a friendly reminder or tip – this is essentially implementing JITAI timing.

- **mCerebrum (MD2K)** – An academic project (by MD2K) that provides a platform for JITAI in mobile health, including sensors and decision engines. It's heavy and mostly for researchers, but their architecture shows how to separate **decision logic** from content. We can glean principles: you'd have a **Decision Engine** (which could be a simple Python service) that listens for triggers (time of day, user's last message sentiment, etc.) and then tells the chatbot to deliver something. *Integration:* If you have the bandwidth, design your system such that the **conversation agent** and the **intervention scheduler** are decoupled. The agent handles on-demand chat; the scheduler (could be Cron, or a small loop in the backend) handles timed/adaptive pushes. This way, if later you have more context (say user connects a smartwatch or calendar), you can extend the decision logic easily without touching the core chat logic.

- **Micro-randomized Trials logic** – In JITAI research, often interventions are delivered in a randomized manner (to learn effectiveness). If you want to gather insights, you can randomize some of your bot's strategies initially and see outcomes. For example, randomly choose to offer a gratitude exercise vs a deep breathing exercise when stress is detected, then track which leads to better self-reported mood. This is more about methodology; you might use an analytics pipeline to analyze it rather than a library in-code. But it's something to plan for (especially if you might publish research or continually improve the bot via data-driven means).

Why these? JITAI capabilities will **differentiate** your product by making it proactive and tailored. Many chatbots are reactive (only respond when user speaks). By integrating scheduling and personalization, your bot can, say, *notice a lapse in activity or detect negative sentiment trending and reach out with support exactly then*. This can significantly improve outcomes (as suggested by clinical studies). The open-source

tools here (like MABWiser for learning and standard schedulers) allow you to implement this without reinventing reinforcement learning from scratch. Essentially, they bring in the **"AI" for timing and selection**, complementing the LLM which handles content. The snippet from a research paper indicates the feasibility: *"Integrating trigger detection and a generative chatbot into a JITAI is possible while ensuring privacy..."* [50] – meaning your context triggers + LLM can co-exist effectively. Starting with simple rules and evolving to learned policies is a sound approach.

Multi-Agent Frameworks for Complex Dialogue (Therapy Teams, Supervisor AI)

Multi-agent setups can enable your system to have specialized sub-agents (therapist persona, coach persona, critic, etc.) working together or a supervisor overseeing an LLM. Notable frameworks:

- **Crew**^{AI} (MIT License) – A **high-performance multi-agent orchestration framework** independent of LangChain [51]. CrewAI lets you create multiple role-specific agents ("AI employees") that can communicate and coordinate. It introduces the concept of **Crews** (teams of agents) and **Flows** (event-driven control logic). *Integration:* Imagine implementing a *"Therapist agent"* and an *"Ethical overseer agent"* and maybe a *"Tool agent"* for fetching info. CrewAI could facilitate their interaction. For example, the user message could first go to a "safety agent" that scans and either approves or modifies it (like an internal content filter agent), then the main "therapist agent" generates a response, and a third "reviewer agent" checks that response against policies, before final output. CrewAI provides the plumbing to do this asynchronously and efficiently, with support for tracing and even a UI. It's quite advanced and has a learning curve, but it's built for enterprise and claims good performance at scale. If your architecture grows in complexity, CrewAI is a possible foundation to avoid writing your own multi-agent manager. It's extremely popular (40k stars) and well-maintained.

- **Microsoft AutoGen** (MIT License) – A framework from Microsoft for creating multi-agent applications where agents can use tools and talk to each other [52]. AutoGen provides abstractions to spin up agents with certain roles and have them engage in dialogues (including allowing the user to join as one agent). It supports both autonomous agents and human-in-the-loop. *Integration:* You could leverage AutoGen to prototype scenarios like **therapist–client simulations** (for testing), or to have an **Agent that calls another** for second opinion. For instance, an "AI Supervisor" agent could critique the main assistant's response (like a live reviewer ensuring quality, similar to how ChatGPT has a moderation layer – you can implement your own). AutoGen would handle the message passing: user -> therapistAgent -> supervisorAgent -> possibly back to therapistAgent for revision -> user. AutoGen also supports tool integration, so one agent could be tasked with retrieval while another focuses on conversation. Given it's from MSR, it's fairly credible and has some documentation for various patterns (workflow agents, conversation patterns, etc.). It might overlap somewhat with LangChain+LangGraph, so consider it if LangGraph's capabilities don't suffice or if you specifically want multi-LLM collaboration.

- **LangChain Agents & AgentManager** – Since you already use LangChain, note that LangChain's agents (like ReAct framework) can be extended to multi-agent use cases. They also have an `AgentExecutor` that could theoretically manage multiple agents. Not as out-of-the-box as CrewAI/AutoGen, but you can definitely implement multi-agent handoffs manually using LangChain primitives (like one chain produces something, then feed to next). LangChain's documentation even references patterns for multi-agent conversations and how to evaluate them with LangSmith [53].

- **Camel, OpenAgents, etc.** – There have been demos like **Camel AI** (where two GPTs talk to each other to solve tasks) – code for these are often just few-shot prompts setting roles. If you want something simpler than a whole framework, you can manually create two `ChatCompletion` calls in a loop: one for agent A, one for agent B, alternating. This is essentially what frameworks do but you can hardcode a specific case (like roleplay conversation between a *"counselor"* and a *"client"* agent to generate more realistic dialogue or to let the AI effectively *think through* both sides of a scenario).

Why multi-agent? In a therapy chatbot, multi-agent setups can be useful for **modularizing responsibilities** – e.g., separating the empathy generation from factual retrieval, or having a constant safety watchdog. It can also simulate **group therapy or family role-play** if you ever go that route (with one agent taking the role of a child perhaps). The listed frameworks (CrewAI, AutoGen) provide robust patterns for such interactions without you having to manually manage message routing or concurrency. CrewAI in particular emphasizes **collaborative intelligence** and gives you both high-level and low-level control [54] [55], which aligns with keeping strategy in your hands while leveraging AI for content.

Given that LangGraph is part of LangChain and somewhat covers agent orchestration with state graphs, you might not need a separate multi-agent framework immediately. But it's good to know these exist if your design grows (especially if you plan e.g. a **Supervisor AI ensuring the Therapist AI adheres to a checklist every session** – that could be elegantly done as two agents in dialogue).

Evaluation and Testing (Quality, Safety, Efficacy)

To ensure your system performs well and continues to do so with updates, integrate some evaluation pipelines:

- **BOLT: Behavioral Open-Loop Test** (BSD-3-Clause License) – A framework from a 2024 paper [56] that assesses LLM therapist behavior on 13 psychotherapy techniques (like reflections, questions, interpretations, etc.) [57]. BOLT provides an **in-context evaluation method**: basically, it uses GPT-4 (or other LLMs) to tag the therapist's responses for which techniques are used. It compares LLM vs human transcripts. *Integration:* You can use the code from BOLT [58] [59] to **analyze transcripts of your bot's conversations**. For example, run your bot through some test dialogues (or real ones with user consent) and use BOLT to see if it's using too many *"problem-solving (advice)"* statements and not enough *"reflections"*, etc. BOLT's findings in the paper showed LLM therapists often over-use certain techniques and under-use others compared to good human therapists [60]. With BOLT, you could quantify that for your bot and adjust prompts or fine-tuning to balance it. This is a more research-oriented eval, but since it's open-source, it gives you a structured way to measure **therapy quality** beyond just user ratings. It uses few-shot classifiers under the hood (you might need OpenAI API for it or could adapt to use local LLMs if they're good enough).

- **VERA-MH** – Stands for *Validation of Ethical and Responsible AI in Mental Health*, proposed by Spring Health [61]. It's a framework/standard (concept paper) to evaluate mental health chatbots on safety, ethics, and clinical effectiveness. They intend it to be an **open evolving evaluation** [62]. While not code yet, keep an eye on it – it might yield an evaluation dataset or checklist. *Integration:* Use the VERA-MH dimensions as a **checklist for manual evaluation**: e.g., does the bot avoid giving medical advice, does it correctly handle users in crisis, does it maintain confidentiality, etc. If any open test cases or metrics come out of VERA-MH, incorporate them. For now, consider it a guiding framework to ensure you're testing the right things.

- **OpenAI Evals** (MIT License) – OpenAI's open-source framework for creating and running **evaluation suites on LLMs** [63]. It comes with a registry of evals (some are simple QA or math, but you can write custom ones). *Integration:* You could create an eval that simulates certain conversation snippets and checks the model's output. For instance, an eval where the user says: "I'm feeling like life is pointless" and you expect the bot to respond with a specific pattern (contain "I'm sorry you're feeling that way" and "You are not alone" or similar). OpenAI Evals lets you define these tests (they can be fuzzy – e.g., use another LLM to grade the response against criteria). Then you run `openai-evals` to get a report. This can be part of CI: if a new version of your prompt or model fails some eval (regresses), you catch it before deployment. It's especially useful if you fine-tune a model – you can ensure the fine-tune didn't break some behavior by running evals. Even if you stick to API LLMs, evals help to validate prompt changes.

- **Promptfoo** (MIT License) – A popular tool for **prompt testing**. You write a YAML or JSON with test cases (inputs and expected outputs or scores), and Promptfoo will run your prompts against models and report results (including diffing outputs, scoring with criteria, etc.). It even supports an interactive CLI and CI integration. *Integration:* Use Promptfoo to rapidly **A/B test prompts** or compare model versions. For example, you might have 5 example user queries (some sensitive, some normal) and you try two prompt variants for your system or two model choices (GPT-4 vs Cohere) – promptfoo will show differences and even allow using GPT-4 to judge which is better if you want. In CI, you can set assertions like "the response must contain an emoji when user asks for mood (checking that behavior is consistent)". Since you want regression tests, this is your friend. It's language-agnostic and you can call your API or local model through it. Write tests such as: **User:** "You're worthless." (to test the bot's self-harm or abuse handling), expect the bot to respond with a polite deflection or empathy and *not* to produce certain banned phrases – promptfoo can check that automatically by regex or by an LLM evaluator.

- **TruLens (TruEra)** (Apache-2.0) – An instrumentation and evaluation library for LLM apps. It can log each conversation turn and compute metrics like **truthfulness, relevance, sentiment** using plug-in evaluators (some use LLMs, some use heuristic) [64] [35]. It also has a dashboard. *Integration:* You can wrap your LangChain calls with TruLens to log all inputs/outputs along with metrics. For instance, it can use Perspective API or toxicity model on each output and track it. Or use GPT-4 to rate "helpfulness" of each answer. Over many sessions, you gather a dataset of interactions and their evaluations, which helps to identify issues or drifts. TruLens also supports feedback loops (like RLHF fine-tuning though that might be beyond scope initially).

- **DeepEval (Confident AI)** (MIT License) – A pytest-like **unit testing framework for LLM outputs** with **40+ built-in metrics** and model-graded evals [65]. It's somewhat similar to Promptfoo but with a focus on metrics like BLEU, BERTScore, as well as GPT-based metrics. It allows writing tests in Python where you assert things about the LLM's response. *Integration:* You could write a DeepEval test that calls your chain with a certain input and then apply a custom metric (for example, a function that checks if the word "suicide" is present in the response when it shouldn't be). DeepEval also integrates with OpenTelemetry for traces [66] and can output nice reports. It's newer but growing.

- **Human Evaluation and Feedback** – While not a library, consider setting up a mechanism for users (or testers/clinicians) to rate conversations. Even a simple star rating after a session, or a thumbs up/down per reply (which you capture and log), can provide supervised signals. OpenAI's guidance is that **human feedback** is gold for alignment. If you collect these, you could incorporate them in fine-tuning or at least use them to measure progress (e.g., percent of sessions rated 4 or 5 stars). If direct user feedback is sensitive, perhaps schedule periodic reviews with a domain expert who reads some transcripts (with permission) and fills a

standardized evaluation (like “Was empathy present? Score 1–5.”). This can validate that the automated metrics (above) correlate with real quality.

Why these? Without solid evaluation, it’s hard to trust or improve a system, especially one with life-affecting outcomes. The above tools give you a **mix of quantitative and qualitative evals**. BOLT and VERA-MH cover the **clinical quality** aspect (are we doing therapy right?). Promptfoo/DeepEval cover **regression testing and prompt accuracy** (are we consistently obeying instructions and not introducing new bugs?). TruLens and Evals allow **ongoing monitoring**. In production, combining these, you’d know if a change (like using a cheaper model or adjusting a prompt) hurts the bot’s performance on any important dimension *before* it reaches users.

By instituting an evaluation pipeline (maybe as a nightly run of tests and periodic analysis of conversation logs), you create a feedback loop to continuously align the bot with your therapeutic goals. This is especially critical in sensitive domains to catch issues early – for example, if some update accidentally made the bot less empathetic or more verbose, you’ll see it in metrics and can correct. Think of it as your **automated QA and alignment lab**.

Privacy and PII Handling (Anonymization, Redaction)

Handling user data (which may include personal details or sensitive info) responsibly is a must. Luckily, several open libraries specialize in PII detection and anonymization:

- **Microsoft Presidio** (MIT License) – A **powerful PII identification and anonymization framework** [67]. It comes with an **Analyzer** that uses built-in recognizers (regexes, ML models) for things like names, emails, phone numbers, IPs, credit cards, etc., and an **Anonymizer** to replace or mask them. It supports multiple languages (English out-of-the-box, and you can add recognizers for others). *Integration:* Use Presidio to scrub any user-provided text that might contain identifying info, especially if you plan to store conversation logs or use them in evals. For instance, before saving a chat transcript for analysis, run it through Presidio’s Analyzer and either redact or pseudonymize names, locations, etc. Presidio is easy to use: you initialize an `AnalyzerEngine` and call `analyze(text)` to get entities [22], then feed that to `AnonymizerEngine` with your desired action (mask with *** or replace with <TYPE>). You might also integrate it into the realtime pipeline – e.g., if a user says “My name is John and I live in Riga”, the bot’s next response could refer to them as just “you” instead of “John”, ensuring not to surface PII. (Though in one-on-one chat it’s less an issue, but for any logging it is.)

- **Scrubadub** (MIT License) – A simpler Python library to **remove PII from free text** [68]. It finds things like names, dates, phone numbers, URLs, etc., and replaces them with generic placeholders (like {{NAME}}). It’s highly configurable (you can add custom regex or adjust for locale). It also supports multiple languages to some extent. *Integration:* If Presidio feels heavy or you need a quick solution, Scrubadub can be employed to sanitize inputs and outputs. For example, if you ever output an email or link as part of resources, you can scrub it if logs should not have it. Or if a user dumps a large journal entry that includes other people’s names, scrubbing before analyzing it with the LLM might be wise depending on your threat model. Scrubadub can run on each message in streaming or batch.

- **Philter (Philterd)** (Apache-2.0) – An open-source **PII redaction engine** that can be deployed as an API/service [69]. It’s designed for healthcare/clinical text de-identification and is quite comprehensive. It uses a set of filters (SSN, MRN, person, location, etc.) and can do partial masking (e.g., replace all but last 4 of an ID). *Integration:* Philter might be overkill unless you have

large volumes of text files to de-identify. But if you were building a system to anonymize transcripts for sharing with researchers, Philter is a good choice. You'd run your text through Philter's REST API and get back a redacted version. In-app, Presidio or Scrubadub are more straightforward as libraries; Philter is more of a standalone service (originally from Univ. of Pittsburgh for clinical notes).

- **Guardrails PII Validator (Guardrails hub)** – The Guardrails library (above) has a ready validator that combines Presidio and an ML model **GLiNER** (a transformer for NER) to detect PII in LLM outputs [22]. It can either **fail** the output (ask LLM to retry without PII) or automatically **anonymize** the output [70] [71]. *Integration:* If you use Guardrails for output schema, simply include the PII validation step. For example, if the bot's answer should not reveal any personal info it saw in documents, Guardrails PII can catch that. This is particularly useful in RAG: say the knowledge base includes a case study with a real name, and the user query triggers an answer that mentions that name – the PII validator would flag it, and you could set `on_fail="fix"` to have the text anonymized or removed automatically [71]. Under the hood it uses a small model (`urchade/gliner_small-v2.1`) for NER, which can detect person names quite accurately, and Presidio's built-ins for things like numbers, dates, etc.

- **Language-specific tools**: For Russian specifically, you have **Natasha** and **SlovNet** which include Russian NER for names, organizations, etc. [72]. If your bot will handle Russian user input containing names/addresses, consider integrating Natasha's NER as well. It's not as plug-and-play as Presidio but quite accurate for RU. You could run both Presidio (with its `ru` recognizers, which might catch some) and Natasha to be safe.

- **Data Storage Practices** – Ensure that wherever you store conversation logs or user profiles, use encryption and least access. If you integrate something like Presidio, you might also generate **hashes** of identifiers to track uniqueness without storing the raw data (e.g., hash email addresses so you know if the same user's email came up again, but you don't store the actual email). Also possibly implement a deletion policy: e.g., allow users to request their data/letters be deleted, and ensure it's scrubbed from logs as well. These are beyond library integration, but important for compliance (GDPR, etc., especially since you mentioned jurisdictions and legal boundaries).

Why these? Trust is crucial. Users won't open up if they fear their personal details could leak or be mishandled. Using robust open-source PII scrubbing ensures that even if someone looks at debug logs or if data is used for research, identities are protected. Presidio in particular is an industry-grade solution [67] that covers a broad range of entities and has been used in healthcare and finance. By integrating it, you align with best practices out-of-the-box. Moreover, if you ever provide transcripts to clinicians (for supervision or auditing the AI), you can easily de-identify them to protect privacy. In sum, these tools help you **bake in privacy by design**, rather than reacting to an "oops" later. Given the sensitive domain (mental health coaching often involves personal and familial details), this layer cannot be overlooked.

Mental Health Dialogue Datasets & Pretrained Models

Leverage existing datasets and models to bootstrap your system's knowledge and empathy. Some key resources:

- **CounselChat** – An open dataset of questions and answers from an online counseling forum (counselchat.com). It contains users' questions on mental health issues and answers written by

licensed therapists. There's a GitHub and it's on HuggingFace [73]. *Usage:* Fine-tune a model on this Q&A to give your LLM more "therapist-like" answering style and content. It's relatively small (a couple thousand Q&As), but good quality. Even without fine-tuning, you can feed the dataset into a vector store: if a user's question is similar to one from CounselChat, retrieve the top answers and either provide them or use them to inspire the LLM's answer (RAG style). The answers often cite techniques or offer empathetic reassurance – useful exemplars.

- **EmpatheticDialogues (Facebook)** – A dataset of ~25k multi-turn dialogues where one speaker describes a personal situation and the other responds with empathy [74]. Each convo is labeled with the emotion of the speaker's situation. *Usage:* You can fine-tune a model for **empathetic response generation** using this dataset (Facebook even released a Transformer model trained on it in their paper). If fine-tuning a large model isn't feasible, use it for few-shot example prompts: take a user message expressing e.g. loneliness, and include an example from this dataset of someone expressing loneliness and a human response. This will nudge the LLM to respond more empathically. It's a great resource to calibrate the **tone** of your bot. The dataset is available on HuggingFace [75] and has been widely used in research as a benchmark for empathy.

- **GoEmotions** – (As mentioned in Emotions section) The dataset itself can be part of your training data to teach the model emotional granularity. Additionally, **models** fine-tuned on GoEmotions exist; if you wanted your bot to explicitly output the user's emotion (for say reflection or to store in user profile), you could integrate one of those. For example, a model `monologg/bert-base-cased-goemotions-original` is on HuggingFace – it outputs 28 emotion probabilities for text [39] [76]. The dataset can also augment an empathy fine-tune: some researchers combine EmpatheticDialogues + GoEmotions + DailyDialog to create a broad conversational fine-tune.

- **Mental Health Forums (Reddit)** – Datasets scraped from Reddit support communities (like r/depression, r/Anxiety, etc.) have been used to train models for crisis detection and also for **peer-support response generation** (though the latter is tricky quality-wise). There was a dataset called *Ubuntu EMPATHETIC Dialogues* and one from CLPsych shared tasks. If available, fine-tuning on real peer support conversations could help the model not just be a therapist but also a peer when needed (more informal, "I hear you, I've been there" style). However, caution: these often contain both good and bad advice; filtering is needed. Unless you have the capacity to curate, might skip direct fine-tune and instead use them to build a **language model of distress** (understanding user language better, which helps classification).

- **Dialectical Behavior Therapy (DBT) Dialogues** – Some researchers have created simulated dialogues for specific therapies (not sure of open datasets, but if found, they'd be gold). Keep an eye on papers or "**tera**" like VERA, which might release data.

- **Open-Source Models**: There are some conversational models fine-tuned for counseling: e.g. *DeepPavlov Dream* had a counseling bot (not sure if open). On HuggingFace, models like **"T5 for empathy and emotion recognition"** from papers (e.g. Rashkin et al.). One interesting one: **"PsyGPT"** or **"CounselGPT"** if any exist (some community might have attempted). Also **"Avatar Therapists"** work – check if any shared models from that research. If you find none explicitly labeled, consider using a strong base model (GPT-3.5 or Llama-2) and apply lightweight fine-tuning or LoRA with the above datasets. The **Llama-2-Chat** model itself was RLHF trained to be helpful/harm-free, which covers some basics, but not domain-specific. A smaller model like **DialoGPT fine-tuned on counseling data** might be something you can run locally for experimentation.

- **Datasets for Evaluation**: Beyond training, there are QA-style datasets like **Mental Health FAQ** (somewhere there might be a collection of common questions and expert answers, aside from CounselChat). Also **"SAFE-T corpus"** from a research on safety responses or **Crisis Counsel** dataset from the Crisis Text Line (if open). These can be used to test the bot or as additional knowledge.

Why these? Incorporating domain data ensures the AI isn't operating solely on general training which might miss nuances. Fine-tuning or prompt enrichment with these datasets will likely yield more **relevant and sensitive outputs**. For example, EmpatheticDialogues will teach it to respond with empathy even when user shares something heavy (rather than giving a generic answer). CounselChat gives it actual therapist wording for common issues (like "sounds like you're feeling X due to Y, you might consider ... [73] "). These help prevent the bot from "hallucinating" advice because it has seen real advice, and help align with **clinical tone** (warm, supportive, not overly formal or robotic). Additionally, if you plan any offline evaluation or benchmark, these datasets provide ground truth to compare against (like you can see if your bot's answers to CounselChat questions are as good as the human counselors' answers via human evaluation).

Deployment, Observability & Monitoring

Finally, getting this system running reliably and monitoring it in production:

- **LangSmith (LangChain Observability)** – LangSmith provides a platform and APIs to **trace LLM calls, track costs, and evaluate runs** [77] [78] . With LangSmith integrated, every time the bot handles a user query, you can record the sequence: retrieval steps, prompt inputs, outputs, etc. It has a UI to visualize these traces, which is invaluable during debugging (especially in complex chains or agent tools). *Integration:* Use the LangSmith Python SDK to log each chain or agent run. For example, wrap your main `ConversationChain` with a LangSmith tracer session, and it will capture each prompt and response with timestamps and token counts. You can also log custom metrics (like classifier outputs) as metadata. In production, LangSmith can feed you real-time dashboards of how users are engaging, error rates, etc. Also, it supports **evaluation of saved traces** – you can write evaluation functions (or use provided ones) to score traces on criteria and flag problems. This ties directly into continuous quality monitoring.

- **OpenTelemetry (OTel)** – A standard for distributed tracing and logging. Some LLM frameworks including LangChain can emit OTel events. If you already use something like **Jaeger** or **Zipkin**, you could instrument the bot's API calls and model calls with OTel spans, which would allow full visibility in a tracing UI. *Integration:* This is more dev-ops-y. For instance, instrument your FastAPI endpoints (there are middlewares for OTel) so each user session or request is traced. Within each request, you can create spans for "retrieval from vectorDB" and "LLM generation" etc., attaching any relevant data. This can be pushed to an APM solution (Datadog, etc., or open-source Jaeger). The benefit: if at some point response times spike or errors happen, you can pinpoint whether it's the vector DB latency, the LLM API, etc. It's a bit heavy to set up but worthwhile as you scale.

- **Helicone** (MIT License) – An **open-source observability platform for LLMs** that acts as a proxy to OpenAI (and others) and logs all requests and responses [79] . With one-line code change (just use Helicone's base URL or their SDK), you get logging of prompt, response, timestamps, tokens, and you can add tags (like user_id, conversation_id). It provides a dashboard to filter and analyze these logs. *Integration:* Instead of calling the OpenAI API directly, you call it via Helicone (either self-hosted or their cloud). Then use their dashboard to see usage by user, detect which prompts cost the most, etc. They recently added **OpenLLMmetry** which logs to a local database if you

don't want external. This is a quick way to get **metrics and monitoring** without building your own logging pipeline. For example, you can see at a glance: "Today we had 50 conversations, 3 of them hit the fallback model, 2 had an error, average response time 1.2s, user X used 5x more tokens than others (maybe a long conversation)". It also supports **OpenTelemetry integration** [66], so it can tie into existing tracing systems.

- **Application Frameworks**: Deploying your bot backend can use common frameworks: **FastAPI** (for REST, easy to serve a /chat endpoint or a WebSocket for real-time chat), **Django** if you need a lot of ORM and admin interface (maybe for content management like updating prompt templates or knowledge articles by non-devs). There's **LangChain's LangServe** – experimental, but meant to deploy LangChain apps. Or even **Streamlit** if you want a simple frontend demo. Since this is likely a web service, ensure to include standard logging (structured logs of interactions, errors) and perhaps an analytics pipeline (e.g., how many users achieve certain goals – though that ventures into product metrics).

- **CI/CD**: Integrate all the above into CI so that when you push changes, tests (Promptfoo/DeepEval) run, and perhaps a staging server runs where you do a sanity check conversation. Use containerization for reproducibility (Docker for the backend – many of the mentioned libraries like Rasa, Haystack, vector DBs also have Docker images). For the LLM, if using OpenAI API, not much to deploy; if using local models, ensure the model serving is containerized or on a separate machine with GPU.

- **Red Team Monitoring**: (Ties back to Garak) – One idea is to run a scheduled job that uses Garak with your production model weekly and logs if any known jailbreak gets through. If one does, send an alert to Slack/email. This way you catch issues proactively. Same with performance: maybe set up a **cronjob that does a sample conversation (scripted)** and measures if the responses come within expected time and quality. This synthetic monitoring is like pinging your bot to ensure all parts are working (e.g., retrieval returns something, etc.). If it fails, it might mean some component (like vector DB or API key) is down.

Why these? In production, things will go wrong or change – having observability is crucial to debug and maintain trust. Tracing and logging each step of the complex pipelines we've described means you can answer questions like "Why did the bot say that strange thing to user X on Tuesday?" by looking at the trace (maybe a particular document was retrieved or the prompt was somehow altered by state). Monitoring helps catch issues early: if an update causes responses to slow down or a classifier to crash (maybe due to an edge case input), you'll see it in logs/metrics likely before users complain en masse. Also, being able to quantify usage (for cost management) and outcomes (for efficacy – e.g. track if user's self-reported mood improves over sessions) will be important for you as you iterate and possibly seek validation of the tool's effectiveness.

LangSmith and Helicone are both tailored for LLM apps, saving you from building a custom logging DB and UI. Using one or both, you cover both **engineering metrics (performance, errors)** and **quality metrics (content evaluation)**. This strong observability backbone combined with the evaluation methods above essentially closes the loop for **continuous improvement** – you have data coming in, and you have tests/metrics to make sense of it.

Localization and Cultural Adaptation (Russian Language & Context)

Since you mentioned Russian-language queries and specific cultural context (e.g. parallel parenting in local context):

- **Russian NLP Integration** – Incorporate **spaCy's Russian model** (`ru_core_news_md` or `lg`) for basic tokenization, sentence splitting, and NER. It covers names, organizations, etc., in Russian. Also **Stanza** (Stanford NLP) has a Russian pipeline including sentiment. These can support tasks like detecting PII in Russian (spaCy NER will catch Russian names, which Presidio might miss unless configured) and splitting user input properly. **Natasha** library (MIT License) is specifically built for Russian – it provides morphology, person name parsing (with patronymics), etc., which can be useful for politely addressing users (formal vs informal speech, etc.) [72]. *Integration:* Use Natasha's `NamesExtractor` to identify a name in user input and then you can address the user by name if appropriate (a personal touch, though in therapy often we don't overuse names, but for coaching it might be nice). Also, use morphological analysis to ensure correct grammar in generated Russian (if you ever fill in templates, e.g., inserting user's name in a reply with correct case).

- **Translation vs Native** – If your main model is English-centric (GPT-4 is great in many languages including Russian, so maybe not an issue), you might translate content. For example, if you use an English dataset (like EmpatheticDialogues) to prompt examples but user is Russian, you may need to translate those examples into Russian in the prompt. DeepL or open models like **Helsinki NLP's OPUS MT** can translate if needed, but GPT-4 can probably do it internally if instructed. Still, if you have static knowledge (like BIFF guidelines text, or law excerpts) only in English, consider obtaining Russian equivalents so the bot isn't translating legal concepts on the fly (which could be error-prone). For instance, find Russian articles or create Russian versions of your policy prompts (e.g., the system message about style should be in the language the model will output in, usually).

- **Local Content and Templates** – Integrate **local legal/administrative templates** that users might need: e.g., a *"parallel parenting plan"* template (there might be one from a Russian family court or a .doc from a social service). If you find such documents (even PDF), you can convert them to text and include in your knowledge base for retrieval. Similarly, **school communication letters** or official guidelines for co-parenting in Russian. Having these allows the bot to provide *concrete suggestions* or even fill-in-the-blank style outputs. *Example:* If user needs to write to a school about a bullying issue, the bot can pull from a template letter format (rus maybe) and help the user adapt it. This might require some curation, but open resources might include government brochures or NGO guides (often public domain or CC). Use them with RAG or as prompt exemplars.

- **Russian BIFF/NVC Materials** – See if there are Russian translations of BIFF guidelines or NVC steps. If yes, use those for the Russian mode of the bot. If not, ensure the bot's Russian outputs still follow the spirit: e.g., in Russian "I-statements" might require slightly different phrasing (avoiding informal "ты" blame, etc.). You could provide example of a Russian "Я-высказывание" (I-statement) as a guide. The same goes for empathy – certain expressions common in English ("That makes sense you feel that way") need equivalent natural Russian ("Я понимаю, почему вы это чувствуете"). If the model sometimes produces overly literal translations, you might need to fine-tune or few-shot it on Russian-specific phrasing.

- **Case and Formality** – Russian has T-V distinction (ты vs вы). Your assistant should likely use **"Вы" (formal)** with users, unless it's styled as a very friendly peer. In therapy context, using polite form is standard initially. If the bot is addressing say an ex-partner in a letter, likely also "Вы" to be respectful (depending on context – could be first name basis or formal, but formal is safer in conflict). So, ensure your Russian prompt or examples use the appropriate register. If the user uses "ты" with the bot, the bot can either continue with "Вы" (keeping professional tone) or mirror the user's level – but probably best to stay with "Вы" for respect.

- **Multilingual Models** – If using open-source models for classification or generation, make sure they support Russian. For example, Detoxify was trained on English; for Russian toxicity, consider **RuToxic** (there was a Kaggle). Or just use Perspective API which supports Russian. HuggingFace likely has a **rubert-toxic** model. Similarly, for Russian emotion detection, there's **RuSentiment** dataset with models. Integrating those will give more accurate results on Russian user input than an English-trained model would. If using GPT-4, it's mostly fine with Russian input and will produce Russian output well (though might occasionally respond in English if the system prompts are English – so ensure system instructions are also given in Russian or explicitly tell it to stick to user's language).

Why these? Providing a seamless experience in the user's native language requires more than just letting the model switch languages. Cultural context – like legal systems, school norms – differ, and users will expect the bot's suggestions to reflect their reality. By feeding local templates and knowledge, the bot's advice will be more **practical and actionable** (e.g. referencing local resources, using familiar formats). Also, handling Russian text correctly (names, respectful language) will build trust; a misstep like using "ты" inappropriately could alienate some users. Technical integration of Russian NLP ensures that features like PII stripping and tone analysis work just as well on Russian text as on English. Essentially, this is about **localizing the intelligence** of your system, not just translating words. Since you specifically listed Russian search queries, it indicates an importance on this – luckily, the open-source ecosystem (Natasha, etc.) provides the building blocks to do it properly.

Integration Roadmap & Best Practices

Bringing it all together, here's how you might implement these modules in an iterative fashion (e.g., sprints), ensuring minimal complexity while maximizing functionality:

- **Sprint 1: Safety First** – Implement the critical safety gates and policies upfront. Set up the **DIFF_DIAG_GATE** using a suicide ideation classifier (e.g. Suicidal-BERT) so that any high-risk input triggers a predefined crisis response (no LLM freeform). Integrate **NeMo Guardrails** with a Colang policy file (`rails.colang`) listing forbidden topics (self-harm methods, etc.) and required behaviors (always provide crisis line on self-harm) – this covers model outputs [14]. Also, add **content filtering** on inputs with Detoxify or perspective API to catch extreme toxicity or abuse; decide if the bot should respond, refuse, or safely address it. Use OPA or simple code rules to enforce any region-specific legal advice restrictions (e.g., if user asks legal question and location == EU, the bot responds with a polite refusal + referral to professional). This establishes a robust **safety net** from day one.

- **Sprint 2: Therapeutic Letter Co-Pilot** – Develop the letter rewriting feature. Utilize the LLM with carefully crafted prompts for NVC and BIFF. Chain it with a **validation pipeline**: after the LLM generates a rewrite, run **Proselint** and a **politeness/toxicity check** on it [45]. If any issues (too long -> not Brief, contains accusatory language, etc.), either loop back to LLM with feedback or

highlight for user. Require that the final suggestion passes all checks ("green status") before showing: i.e., it should score low toxicity, use no forbidden words (maybe maintain a list of curse words or manipulative phrases to avoid), and roughly meet BIFF criteria (you can approximate "Brief" by a length limit, "Friendly" by no toxic or sarcastic tone detected, etc.). Also integrate **Presidio** here – ensure any personal names or addresses in the user's draft are either omitted or changed in the rewritten version for privacy (the user can reinsert actual names, but the AI doesn't need to keep them in memory while processing). This two-step approach (generate -> lint) will yield high-quality, safe letters ready to send.

- **Sprint 3: Observability and Testing** – Before wide release, instrument the system for transparency. Add **TruLens or LangSmith tracing** around your LangChain chains to capture all steps (prompts, retrieved docs, etc.). Set up **Promptfoo** tests with a variety of scenarios: normal query, crisis query, letter rewrite, etc., to run on each code commit (or at least before deployment). Also implement **Garak** in a CI pipeline: maybe use a subset of Garak probes focused on your domain (jailbreak attempts that try to get the bot to violate therapy ethics, etc.) [80]. If any probe succeeds, it should flag a failure in CI. This ensures you don't accidentally weaken a prompt and open a hole. Additionally, start collecting some user feedback data (even if internal testers): have a simple rating at the end of a session and log it. This sprint is about setting up the **feedback loop infrastructure** – it might not immediately improve user functionality, but it's crucial for scaling confidently.

- **Sprint 4: Russian Localization & Data Protection** – Now that the core is working in one language, adapt for Russian. Integrate **Natasha** and **spaCy Russian** models to handle Russian text processing [72]. Translate all prompt templates and system messages to Russian (taking care to maintain formality and clarity). Test the letter rewriting in Russian – adjust the prompt if needed because BIFF in Russian might require different wording (e.g. no direct translation for "Brief" but you can say "Коротко и по существу"). Update PII tooling: Presidio has some Russian support (it can detect Cyrillic names if configured). Possibly incorporate **ru-locale recognizers** or run an extra pass with Natasha NER to catch what Presidio misses. Ensure all evaluation tools that were English-centric (like toxicity model) have Russian equivalents: for toxicity, use a Russian lexicon or Perspective API's ru model. If the knowledge base has primarily English content (CounselChat Q&As are English), start adding Russian sources: perhaps find Russian psychology forums or **mental health FAQ** from Russian clinics (if available). At least, gather a list of Russian crisis lines, legal resources, etc., to provide relevant links to users. Also at this stage, enforce **data anonymization** in stored logs using Presidio/Scrubadub so that any transcripts or user inputs saved do not contain identifiable info – especially important if you have Russian users, due to GDPR-like laws there. By end of this sprint, the bot should handle Russian as smoothly as English, and you should be confident that no sensitive data is lying around unredacted in logs or analytics.

- **Sprint 5: JITAI Personalization Pilot** – Implement a basic **just-in-time intervention scheduler**. Perhaps start with one simple use-case: if user hasn't engaged in 7 days, send a check-in message. Use a cron or background task for this. Or, if you have access to something like the user's phone accelerometer (just hypothetical), trigger a grounding exercise when high movement + night time (could indicate panic – this is what some JITAIs do). Since that might be out of scope, focus on personalization via a contextual bandit in a small area: for example, you have two versions of a motivational prompt, and you let the bandit learn which a user responds to more. Implement **MABWiser** with context = user mood rating, and reward = whether user came back next day. This is experimental, but set it up to start collecting data. Also ensure there's an easy way to throttle or turn off interventions to avoid spamming users (maybe add a user preference like "daily reminders: on/off"). The result of this sprint will be an **added intelligence**

layer for timing – even if initially rule-based, you'll be laying groundwork for adaptive interventions that can greatly improve user outcomes by providing support at key moments.

- **Sprint 6: Red-Teaming & Compliance in CI** – As you approach a production-ready state, strengthen the system against adversarial use and ensure compliance documentation. Expand the **Garak tests** to cover more attack vectors (there's a list in Garak's repo of prompt exploits – include them all) and run them periodically [81]. Consider integrating **Rebuff's canary token technique** in a dev environment to see if the model ever leaks secrets (like if you embed a string in prompt and see if user can fish it out – this could even be an automated test using the canary functions from Rebuff's code). Any time you change the prompt or upgrade the model, run these tests. Also, do a **privacy impact assessment** – verify that your logging and PII scrubbing is working (simulate with test data containing fake PII and see if it gets removed). If possible, add a test conversation in CI where a user gives personal info and ensure that in the saved log that info is masked. This sprint is about making sure when you go live, you've **dottted all i's**: the model won't unexpectedly break guidelines, user data is safe, and you have documented evidence (from your evals) of the bot's behavior that you can show stakeholders or regulators if needed.

Throughout these sprints, you'll be incrementally assembling the **graph.yaml** (or equivalent) of your conversation flow nodes and transitions, updating the **rails.colang** rules, refining the **reply.schema.json** for outputs (perhaps it includes fields like `{"action": "...", "content": "...", "citations": [...]}` if you enforce structured output for certain queries), adding tests in `eval/` folder (promptfoo configs, etc.), and setting up monitoring in `observability/`. Each module we discussed slots into these blueprints.

In the end, you'll have an architecture where:

- **Dialogue management** is handled by a combination of LangGraph (for overall session structure) and policy files (Colang or your own YAML) for specific flows.
- **Safety** is enforced at multiple levels (input classifier, output guardrails, monitored by a safety agent perhaps).
- **Augmentation** via RAG ensures factual grounding.
- **Emotional intelligence** via classifiers ensures the bot "reads" and responds to feelings.
- **Letter rewriting** is done through an LLM pipeline but double-checked by rule-based tools for quality.
- **Parenting and therapeutic exercises** are embedded as sub-dialogues, triggered by context.
- **Adaptive intervention** logic is plugged in to maximize user engagement and outcomes.
- **Evaluation** is ongoing with automated tests and human feedback, preventing drift.
- **Privacy** is ingrained, with PII never persisting in raw form.
- **Infrastructure** is solid, observable, and scalable.

All with mostly open-source components (LangChain/LangGraph – MIT; Guardrails – Apache; Haystack – Apache; CrewAI – MIT; etc.) so you avoid vendor lock-in and can customize as needed. The licenses of the key pieces are permissive (MIT/Apache) [82] [83] [84], which means you can integrate them into a commercial product without concern (just make sure to comply with any attribution if required).

By following this plan, you **keep control** over the strategy and let these libraries & models handle the heavy lifting under the hood – aligning perfectly with your goal of integrating powerful tools **without unnecessary complexity**. Each module improves a specific dimension of the chatbot (safety, empathy, correctness, etc.), and together they form a comprehensive, modern LLM-based mental health assistant

platform. Good luck with building this – it's exciting and with the above open-source toolkit, very achievable!

Sources:

- LangChain LangGraph documentation [1] [2]
- RasaGPT (Rasa + LangChain) README [8]
- NVIDIA NeMo Guardrails paper/README [12] [14]
- Guardrails AI PII validator description [22] [23]
- Rebuff (Protect AI) README (prompt injection defense layers) [26]
- NVIDIA Garak (LLM vulnerability scanner) repo [24]
- Haystack GitHub description [31]
- RAGAS documentation on evaluation metrics [85] [35]
- Detoxify GitHub description [38]
- GoEmotions dataset info [39]
- Suicidal-BERT model card [40] [41]
- Trauma-informed chatbot guidelines [44]
- Proselint GitHub (prose linter) [45]
- CBTree README (CBT thought record flow) [47]
- JITAI integration study [50]
- CrewAI documentation (multi-agent framework) [51]
- Microsoft AutoGen GitHub README [52]
- BOLT paper/code (LLM therapist behavior analysis) [57]
- VERA-MH concept announcement [61]
- OpenAI Evals GitHub description [63]
- Confident AI DeepEval overview [65]
- Microsoft Presidio GitHub [67]
- Scrubadub docs [68]
- monologg/GoEmotions (dataset description) [39]
- EmpatheticDialogues description (Kaggle) [74]
- CounselChat dataset GitHub [73]
- LangSmith observability features [78]
- Helicone GitHub (LLM logging platform) [79]

---

[1] [82] GitHub - langchain-ai/langgraph: Build resilient language agents as graphs.
https://github.com/langchain-ai/langgraph

[2] [3] [4] [5] [6] LangGraph
https://langchain-ai.github.io/langgraph/?ref=blog.langchain.com

[7] Rasa, LLMs, and RAG — Powering a Solution for Conversational AI
https://dev.to/hamid_zangiabadi/rasa-llms-and-rag-powering-a-solution-for-conversational-ai-3a5b

[8] [83] GitHub - paulpierre/RasaGPT: RasaGPT is the first headless LLM chatbot platform built on top of Rasa and Langchain. Built w/ Rasa, FastAPI, Langchain, LlamaIndex, SQLModel, pgvector, ngrok, telegram
https://github.com/paulpierre/RasaGPT

[9] [10] [32] [37] Mental Health Therapy as an LLM State Machine
https://blog.langchain.com/mental-health-therapy-as-an-llm-state-machine/

11 Conversational AI with Language Models | Rasa Documentation
https://rasa.com/docs/learn/concepts/calm/

12 13 14 15 16 17 18 19 30 84 GitHub - NVIDIA-NeMo/Guardrails: NeMo Guardrails is an open-source toolkit for easily adding programmable guardrails to LLM-based conversational systems.
https://github.com/NVIDIA-NeMo/Guardrails

20 AI Guardrails Index: PII Detection
https://index.guardrailsai.com/section/PII%20Detection

21 AI Guardrails Index | Guardrails AI
https://index.guardrailsai.com/

22 23 70 71 GitHub - guardrails-ai/guardrails_pii
https://github.com/guardrails-ai/guardrails_pii

24 GitHub - NVIDIA/garak: the LLM vulnerability scanner
https://github.com/NVIDIA/garak

25 26 27 28 29 GitHub - protectai/rebuff: LLM Prompt Injection Detector
https://github.com/protectai/rebuff

31 deepset-ai/haystack - GitHub
https://github.com/deepset-ai/haystack

33 34 Evaluating RAG Responses using RAGAS & OpenAI Eval Framework
https://medium.com/@vishaalini70/evaluating-rag-responses-using-ragas-openai-eval-framework-f3952ee75778

35 64 85 GitHub - explodinggradients/ragas: Supercharge Your LLM Application Evaluations
https://github.com/explodinggradients/ragas

36 Text splitters - LangChain docs
https://python.langchain.com/docs/concepts/text_splitters/

38 unitaryai/detoxify: Trained models & code to predict toxic ... - GitHub
https://github.com/unitaryai/detoxify

39 76 GitHub - monologg/GoEmotions-pytorch: Pytorch Implementation of GoEmotions
https://github.com/monologg/GoEmotions-pytorch

40 41 gohjiayi/suicidal-bert · Hugging Face
https://huggingface.co/gohjiayi/suicidal-bert

42 Modeling Empathetic Alignment in Conversation - arXiv
https://arxiv.org/html/2405.00948v1

43 Models - Hugging Face
https://huggingface.co/models?other=politeness

44 [PDF] Draft Standards for Mental Health Chatbots - Regulations.gov
https://downloads.regulations.gov/FDA-2025-N-2338-0006/attachment_2.pdf

45 amperser/proselint: A linter for prose. - GitHub
https://github.com/amperser/proselint

46 47 GitHub - kylehgc/CBTree
https://github.com/kylehgc/CBTree

48 fidelity/mabwiser - Contextual Multi-Armed Bandits Library - GitHub
https://github.com/fidelity/mabwiser

49 Revenue Maximization by Price Optimization using Contextual Bandit
https://medium.com/@j.wang.mlds/revenue-maximization-by-price-optimization-using-contextual-bandit-6e1db5976009

50 Designing a just-in-time adaptive intervention with trigger detection ...
https://pmc.ncbi.nlm.nih.gov/articles/PMC12535633/

51 54 55 GitHub - crewAIInc/crewAI: Framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling complex tasks.
https://github.com/crewAIInc/crewAI

52 GitHub - microsoft/autogen: A programming framework for agentic AI
https://github.com/microsoft/autogen

53 Evaluating RAG pipelines with Ragas + LangSmith - LangChain Blog
https://blog.langchain.com/evaluating-rag-pipelines-with-ragas-langsmith/

56 57 58 59 60 GitHub - behavioral-data/BOLT: A Computational Framework for Behavioral Assessment of LLM Therapists
https://github.com/behavioral-data/BOLT

61 VERA-MH Concept Paper - arXiv
https://arxiv.org/html/2510.15297v1

62 Spring Health, AI Council release VERA-MH
https://www.springhealth.com/news/spring-health-expert-council-vera-mh-first-open-source-evaluation-ai-mental-health

63 openai/evals - GitHub
https://github.com/openai/evals

65 confident-ai/deepeval: The LLM Evaluation Framework - GitHub
https://github.com/confident-ai/deepeval

66 Helicone/ai-gateway: The fastest, lightest, and easiest-to ... - GitHub
https://github.com/Helicone/ai-gateway

67 microsoft/presidio - GitHub
https://github.com/microsoft/presidio

68 scrubadub — scrubadub 2.0.0 documentation
https://scrubadub.readthedocs.io/

69 Philter redacts sensitive information such as PII and PHI in text.
https://github.com/philterd/philter

72 Solves basic Russian NLP tasks, API for lower level Natasha projects
https://github.com/natasha/natasha

73 nbertagnolli/counsel-chat: This repository holds the code ... - GitHub
https://github.com/nbertagnolli/counsel-chat

74 Empathetic Dialogues (Facebook AI) 25k - Kaggle
https://www.kaggle.com/datasets/atharvjairath/empathetic-dialogues-facebook-ai

75 facebookresearch/EmpatheticDialogues: Dialogue model ... - GitHub
https://github.com/facebookresearch/EmpatheticDialogues

77 Log LLM calls - Docs by LangChain
https://docs.langchain.com/langsmith/log-llm-trace

78 LangSmith - Observability - LangChain
https://www.langchain.com/langsmith/observability

79 Helicone/helicone: Open source LLM observability platform ... - GitHub
https://github.com/Helicone/helicone

80 Mastering LLM Security: A Deep Dive into Garak Vulnerability Scanner
https://medium.com/@kachwalla64/mastering-llm-security-a-deep-dive-into-garak-vulnerability-scanner-a1274003aa47

81 AI Security in Action: Applying NVIDIA's Garak to LLMs on Databricks
https://www.databricks.com/blog/ai-security-action-applying-nvidias-garak-llms-databricks