ChatGPT

Решения по модулям для чат-бота поддержки родителя

State Machine (Оркестрация диалога)

- **Apache Burr (incubating)** – фреймворк (Apache 2.0) для построения приложений на основе конечных автоматов (state machines). Позволяет задавать **состояния и переходы** диалога, интегрируя LLM и собственные функции. Включает **UI для мониторинга** выполнения агента и хранение состояния (через БД или файловое хранилище) [1] [2]. **Интеграция:** Python-библиотека (`pip install burr`), совместима с FastAPI (в планах встроенный FastAPI сервер) [3]. Можно использовать для явного моделирования шагов диалога, **отладка** через UI. Пример: реализация чатбота или RAG-ассистента с наглядным **трассированием** в Burr [4].

- **LangGraph** – новый модуль от команды LangChain для **оркестрации workflows с состояниями** (pip `langgraph`). Позволяет определять **граф состояний** и переходов, поддерживает длительное выполнение с сохранением контекста [5] [6]. **Особенности:** хранение памяти беседы между вызовами (собственные `StateGraph` и `MessagesState`), **человеческое вмешательство** по чекпоинтам, стриминг токенов [7] [8]. **Интеграция:** LangGraph – низкоуровневый Python-фреймворк, совместим с LangChain, но может использоваться отдельно [9] [10]. В вашем стеке LangGraph уже упоминается, что облегчает внедрение: можно проектировать диалог как граф (yaml/код) с узлами `DIFF_DIAG_GATE`, `LETTER_COACH` и т.д., определяя логику переходов.

- **LLM State Machine (robocorp)** – легковесная библиотека (Apache 2.0) для создания агентов GPT с явной **логикой состояний** и **памятью чата** [11] [12]. Разрешает только заданные действия для каждого состояния (через генерацию JSON-функций), что обеспечивает **контролируемое поведение** LLM [13]. **Пример:** агент, играющий в мемори, где состояния `INIT` и `COMPLETE` определяют, какие функции (действия) доступны, а история ходов – часть памяти [11] [14]. **Интеграция:** устанавливается `pip install llmstatemachine`. Можно встроить в FastAPI как часть backend-логики, определив состояния: например, `DIFF_DIAG_GATE` вызывает функции классификации профиля, затем переходит в состояние `LETTER_COACH` с разрешёнными действиями генерации письма и т.д. Этот подход совместим с PostgreSQL для хранения истории (история может храниться целиком как контекст, либо в БД по завершении).

Letter Writing (генерация писем: NVC/BIFF)

- **NVC Coach (репозиторий)** – открытый проект (GPL-3.0) для практики **ненасильственного общения (NVC)** [15] [16]. Содержит prompt-инструкции для роли *NVC-тренера*, помогающего переформулировать высказывания в стиле эмпатии и потребностей. Например, реализована функция “do-over”: бот разыгрывает с пользователем конфликтную ситуацию, затем предлагает переписать фразу в духе NVC [17] [18]. **Интеграция:** из репозитория Alexei-Novak/nvc-coach можно взять *подход к промптам* – системное сообщение с правилами (например, всегда идентифицировать чувства/

потребности и предлагать вариант фразы по NVC). Можно адаптировать под ваш `LETTER_COACH` модуль: пользователь вводит черновик письма, а бот возвращает версию, сформулированную в стиле NVC. Также можно учесть **BIFF**: техники краткого, информативного, дружелюбного, твёрдого ответа.

- **BIFF Prompt Template** – готовый шаблон (не библиотека, а инструкция) для генерации ответа в стиле BIFF [19] [20]. Пример системы: *"Проанализируй приведённое конфликтное сообщение. Затем с помощью метода BIFF (Brief, Informative, Friendly, Firm) набросай ответ, который снизит накал конфликта."* Шаблон разбивает задачу на шаги: выделить эмоции, быть кратким, дать факты, сохранить дружелюбный тон и чётко обозначить границы [19] [20]. **Интеграция:** такую инструкцию можно использовать внутри состояния `LETTER_COACH`. Например, if пользователь просит помочь ответить бывшему супругу, система генерирует BIFF-ответ. Можно реализовать как функцию: на вход сырой текст письма, на выходе – отредактированный текст. Для многоязычности можно настроить prompt под нужный язык. *Примечание:* Хотя специализированных готовых библиотек под BIFF/NVC нет, **LLM-модели (GPT-4)** отлично справляются с этими стилями при правильно заданных **системных сообщениях и примерах** [21] [22].

- **Peacy (AI-медиатор)** – свежий проект (MIT) ориентирован на разрешение конфликтов с принципами NVC [23]. Использует стэк: **Python + LangChain + ChromaDB + PostgreSQL** для памяти профиля, **OpenAI модели** для эмпатичных ответов [24]. Peacy хранит профили и сводки бесед в Postgres и использует LangChain для поиска релевантных данных [25]. **Интеграция:** хотя у Peacy мало звёзд (новый проект), его подход близок к нашему: persistent memory (PG), векторное хранилище (Chroma) для диалогового контекста, инструменты (APScheduler для фоновых задач). Можно изучить README и взять **готовые практики**: например, организация профиля пользователя (предпочтения, триггеры) в БД, использование памяти при генерации. Peacy демонстрирует, как **эмпатичный агент** ведёт беседу ненасильственно, что релевантно модулю `LETTER_COACH` и `MEDIATION`.

(Лицензии: NVC Coach – GPL3, Peacy – MIT. Для production лучше обратить внимание на совместимость лицензий.)

Just-In-Time Adaptive Interventions (JITAI)

- **APScheduler / Task Scheduling** – проверенное решение (MIT) для планирования фоновых задач в Python (e.g. отправка интервенций по расписанию). JITAI предполагает предоставление поддержки **в нужный момент**, адаптивно. Например, бот может через APScheduler проверять раз в день эмоциональное состояние пользователя и предлагать упражнение, или реагировать на внешние триггеры (календари, события). **Интеграция:** APScheduler легко встраивается в FastAPI (Background tasks) – можно планировать, скажем, напоминание о выполнении цели или отправку полезной статьи в момент, когда у пользователя стресс. Это позволит модулю `JITAI` динамически подталкивать пользователя к здоровым действиям (Just-in-time советы).

- **Персонализированные триггеры + LLM** – последние исследования показывают, что LLM способны генерировать **адаптивные интервенции** лучше людей [26] [27]. В работе *"The Last JITAI? Exploring LLMs for Issuing JITAIs"* GPT-4 превосходил экспертов по уместности и эффективности советов для пациентов (реабилитация сердца) [28]. Практически это можно применить так: имея данные о пользователе (настроение, события), **модель генерирует микро-интервенцию** (фразу поддержки, упражнение) в нужный момент. **Интеграция:**

модуль `JITAI` может состоять из правила (эвента) + шаблона промпта. Например, если пользователь 3 дня не достигает цели, LLM формирует мотивирующее сообщение в духе МИ (мотивационного интервью). Это потребует сбора контекста (хранится в PG) и вызова модели по расписанию. Для надежности можно начать с фиксированных шаблонов (BIFF/NVC фразы по событию), а затем улучшить LLM-генерацией.

- **MobileCoach / Open mHealth (альтернатива)** – существуют открытые платформы eHealth (например, MobileCoach, PathMate) для JITAI, но они не интегрированы с LLM. Поэтому оптимально **сделать свой адаптер**: хранить правила интервенций в БД (PostgreSQL) и иметь фоновые воркеры. Ваша архитектура (FastAPI + state machine) вполне позволяет это без внешних зависимостей. *Вывод:* JITAI модуль – это комбинация **планировщика задач, триггеров состояния** и генератора сообщений. Production-ready компоненты: APScheduler (в FastAPI), или Celery + RabbitMQ для более тяжёлых задач, можно дополнить библиотекой **pytz** для учета часовых поясов и библиотекой **pendulum** для удобной работы со временем.

Retrieval-Augmented Generation (RAG)

- **Haystack (deepset)** – крупный open-source фреймворк (Apache-2.0) для **поиска знаний и RAG**. Предоставляет конвейеры: коннекторы к базам знаний, преобразование документов, **векторный поиск** и генерация ответа LLM с цитатами [29] [30]. Haystack заточен под **production QA системы**: поддерживает ElasticSearch, FAISS, Weaviate, Milvus и др. для индексации. **Интеграция:** можно запустить Haystack как бэкенд (REST API) или использовать модули внутри FastAPI. К примеру, модуль `RAG` вашего бота может использовать Haystack Pipeline: вопрос -> поиск по базе (например, базе статей о воспитании, законах) -> генерация ответа с цитированием источников. Haystack хорошо документирован [31], есть готовые примеры чатботов с RAG (ваш случай – многоязычный, но Haystack поддерживает и мультиязычный поиск при соответствующих эмбеддингах). Лицензия Apache позволяет без проблем интегрировать.

- **LlamaIndex (GPT Index)** – библиотека (MIT) для интеграции данных в LLM-приложения [32] [33]. Она выступает как **прослойка** между вашими данными и моделью: умеет подключаться к множеству источников (файлы, PDF, базы данных), создавать различные индексы (списки, древовидные, графовые) и **выполнять запросы** с контекстом [33]. Преимущество – модульность: есть готовые коннекторы (LlamaHub) к Google Drive, Notion, SQL и др. [34] [35]. **Интеграция:** LlamaIndex можно использовать вместе с LangChain или отдельно. Например, можно хранить **профиль пользователя и прошлые советы** в индексе, и при новом запросе делать `query_index` чтобы подтянуть релевантные факты (реализация памяти через RAG). Также можно создать индекс по библиотеке статей (развод, школа, медиация и т.п.) и в ответах бота приводить выдержки. Фреймворк поддерживает **Postgres (через pgvector)** и другие vector stores. Он совместим с FastAPI (просто Python), и может работать внутри LangGraph (LangChain), т.к. существует интеграция с LangChain tools.

- **Хранилище данных**: учитывая стек (PostgreSQL), логично использовать **pgvector** для векторного поиска. **pgvector** – расширение PG (сам PG остаётся хранилищем), даёт возможность хранить эмбеддинги и выполнять поиск ближайших соседей. Production-ready: используется в многих проектах, лицензия PostgreSQL. **Интеграция:** установить расширение на вашу PG БД, использовать, напр., **LlamaIndex** или **LangChain Retriever** с `PGVector` backend. Это упростит инфраструктуру (не нужен отдельный Milvus/Chroma-сервер). Если же требуется более масштабно: **ChromaDB** – легковесное встроенное

векторное хранилище (аналоги: Weaviate, Milvus – но они тяжелее). Chroma (MIT) можно запустить локально, он хорошо интегрируется с LangChain. *Резюме:* RAG модуль может быть реализован либо через крупный фреймворк (Haystack) – больше возможностей (построение pipe, ранжирование), либо ручной связкой (LangChain Retriever + PGVector). Оба пути production-проверены [36] [37].

Memory (память диалога и профиля)

- **Memori (OpenLLM Memory Engine)** – новая библиотека (≈1.5k звезд, Apache-2.0) для **долговременной памяти** LLM [38] [39]. Ключевая идея: хранить структурированную память в **SQL-базе** (SQLite, PostgreSQL, MySQL), с извлечением сущностей и отношений из диалогов. Memori автоматически извлекает из беседы **факты, предпочтения, имена** и сохраняет как записи; затем при новых запросах умеет подгружать **релевантные "воспоминания"** [40] [41]. **Интеграция:** одной строчкой `memori.enable()` можно подключить память для OpenAI, LangChain или Anthropic моделей [42]. Memori прекрасно вписывается в ваш стек: используйте PostgreSQL как back-end хранилище памяти (Memori поддерживает PG) [43] [44]. Например, профиль пользователя (возраст, ситуация, цели) будет сохранён как факты; при общении Memori будет автоматически предоставлять модели нужные детали (имена детей, договоренности по опеке и т.п.). Это соответствует требованию **разделения памяти/профиля** – Memori как раз хранит *семантическую* долговременную память отдельно, а *краткосрочную* держит в контексте запроса [45] [46]. Можно настроить режим: **Conscious mode** – краткий контекст (важные факты из памяти добавляются единоразово в системное сообщение), или **Auto mode** – динамический запрос к БД на каждый вход (Memori сама решает, что извлечь) [45] [46].

- **Microsoft Presidio (PII сканер)** – (MIT, ~6k звезд) поможет на этапе памяти **очищать персональные данные** перед сохранением [47]. Presidio обнаруживает номера телефонов, адреса, ФИО и др. чувствительные данные и умеет их замазывать или анонимизировать [47]. **Интеграция:** перед сохранением сообщений пользователя в память можно прогонять через Presidio Analyzer и, например, заменять реальные имена на псевдонимы или хэш. Это важно, чтобы даже во внутреннем хранилище не хранить лишние PII (соответствие модулю privacy). Memori, кстати, тоже поддерживает приватность – хранение на вашем контролируемом SQL и возможность экспорта всей памяти (для проверки) [43] [42]. Таким образом, связка Memori + Presidio даст **прозрачную, контролируемую память**: данные в PG, обезличены и под полным контролем (соответствует trauma-informed privacy требованиям [48] [49]).

- **Redis as Memory (Redis OpenAI Memory)** – альтернативно, **Redis** часто применяется для памяти агентов (пример: LangChain ChatMessageHistory на Redis, либо RedisVectorStore для длинных диалогов). Redis обеспечивает очень быстрый доступ (in-memory), поддерживает векторный поиск (Redis Vector index) и JSON-хранилище. **Production кейсы:** ManyChatGPT-боты используют Redis для хранения контекста между сообщениями. **Интеграция:** можно настроить **Redis Stack** с модулями (RedisSearch, RedisJSON) и использовать готовые реализации memory от LangChain (ConversationBufferMemory с backend=Redis). Однако, с учётом вашего стека, добавление Redis – усложнение. PostgreSQL + Memori уже покрывают хранение. Но можно **скомбинировать**: **PG для долговременной памяти**, **Redis для рабочей** (кратковременной) – например, хранить последние 10 сообщений в Redis для быстрого доступа (или summary), а всё остальное – в PG. RedisLabs опубликовали гайд по разделению short-term vs long-term memory с примерами (summary, vector search, extraction, graph storage) [50] [51]. Это поможет выстроить **многоуровневую память**:

краткая память в контексте (что решается LangGraph checkpoint'ами или Memori conscious mode), долговременная – в БД (Memori DB).

Privacy (Конфиденциальность и PII)

- **Microsoft Presidio** – полный стек для обнаружения и удаления PII в тексте (и даже на изображениях). Содержит каталог детекторов (номер карты, email, имя, паспорт и т.п.) на основе шаблонов, NLP-моделей и словарей [47]. **Применение:** на этапе ввода пользователя **анализировать сообщения** Presidio Analyzer'ом и помечать чувствительные места. Далее, **Presidio Anonymizer** может заменять их на <MASK> или другими токенами. Это важно, чтобы: (1) **не хранить лишнее** в логах/памяти; (2) **модель не утекала личные данные** при ответе. Например, если пользователь назвал имя ребёнка, можно в память сохранить только "имя=И.". Presidio легко встраивается (Python API), работает локально (никаких внешних API – важно для privacy!). Лицензия MIT разрешает коммерческое использование [52].

- **Непосредственное хранение профиля отдельно** – уже архитектурно правильно, что профиль/анкета хранится отдельно от истории диалога. Следует обеспечить, что при генерации ответов LLM не раскрывает больше, чем нужно. Можно добавить **политики Guardrails** (см. ниже Safety) – например, запретить боту сообщать конкретные адреса/телефоны даже если они есть. **Травма-ориентированная приватность:** согласно рекомендациям исследований, система должна обеспечивать анонимность в реальном времени и избегать хранения идентифицирующих данных, чтобы пользователь чувствовал себя в безопасности [53] [49]. Инструменты: Presidio (PII удаление), spaCy NER (определять сущности и, например, заменять на категории – "<имя_ребёнка>"). В *MDPI 2023 (PTSD AI)* архитектуре предлагается **реальный time anonymization** на входе, до подачи текста в модель [48] [49] – мы можем сделать так же, подключив Presidio как первый шаг обработки запроса.

- **Диалог без камер и микрофонов** – (Приватность не только PII, но и комфорт пользователя). Упомянем: некоторые решения добавляют "псевдо-анонимность" – например, **замена голоса** если аудио-чат, или **UI без лишних логов**. В нашем случае текст, поэтому главное – хранение и передача. Использование собственных API (OpenAI) нужно настроить так, чтобы **не логировать содержание запросов** (параметры API). Также стоит добавить **Privacy Policy** явную: чтобы пользователь знал, как хранятся данные. Это не техническая библиотека, но готовые практики – например, шаблон GDPR-совместимого consent можно взять из проекта Wysa или Replika.

(Вывод: модуль Privacy – это скорее процесс: сканировать PII (Presidio), удалять/маскировать, хранить отдельно личные данные, минимизировать их в prompt. Все используемые библиотеки – production-grade и широко применяются в индустрии.)

Trauma-Informed Safety (Безопасность с учётом травмы)

- **NeMo Guardrails + Colang** – специализированный фреймворк (Apache-2.0) для описания **правил поведения бота** [54] [55]. Позволяет в декларативном формате (.colang файлы) задать *rails*: о чём **не говорить**, как реагировать на определённые **триггеры**, когда звать человека. Для травмоориентированного бота важно: **не допускать ретравматизации** и опасных советов. С помощью Guardrails можно, например, запретить обсуждать графичные подробности насилия или суицида, а вместо этого переводить разговор в

безопасное русло (или показывать контакты кризисной помощи). **Интеграция:** NeMo Guardrails работает как прослойка между вашим приложением и LLM: вы определяете rails, и затем запускаете `guardrails.run(prompt)`, получая уже отфильтрованный ответ. Guardrails поддерживает **категории уязвимостей LLM** (джейлбрейки, инъекции) и может автоматически проверять выход на запрещённые темы [56]. С учётом, что вы уже планируете rails.colang, можно написать правила типа: *Когда пользователь сообщает о желании причинить себе вред – ответить по скрипту "Сообщение о поддержке+рекомендация обратиться к специалистам" (и не продолжать обычный диалог)*. Или: *Если пользователь ругает бота/агрессирует – включить EMPATHY_CHECK и отвечать устойчиво, не эскалируя конфликт*. Guardrails позволяет задавать и **стиль языка** ответа на уровне rails (например, `style: compassionate` – указывая, что ответы должны быть в сочувственном тоне).

- **Психологические протоколы в логике** – помимо ML-классификаторов, исследователи рекомендуют внедрять явные **символьные правила** по протоколам травма-терапии [57] [58]. В упомянутой статье про PTSD AI введён *символьный контроллер*, предотвращающий небезопасный вывод: LLM работает под надзором логики, которая **не допускает** нарушений протокола (например, не задавать вопросы, которые могут спровоцировать флэшбэк) [59] [60]. Мы можем реализовать часть этих правил через Guardrails/Colang или в коде LangGraph. Например: *ограничить длительность погружения в травматичные воспоминания* – если бот видит, что пользователь "застрял" в негативе, то запустить модуль `SAFE_COPING` (предложить упражнение на стабилизацию). Такие **"детерминистские"** перила защитят от ошибок нейросети. **Интеграция:** Colang позволяет писать условия на вход/выход, например: `when user says {#intent_crisis} then respond $CRISIS_RESPONSE` – подобное правило уберёт вариативность и обеспечит консистентную реакцию в кризисных ситуациях. Также Guardrails можно расширять Python-хэндлерами: например, при срабатывании правила *суицидальный сигнал* – вызывать функцию, отправляющую уведомление модератору/психологу.

- **Контент-модерация и модели** – параллельно стоит использовать **классификаторы вредного контента**. OpenAI Moderation API (или open-source аналоги вроде **Suicidal-BERT** [61], **HateXplain** и др.) может помечать входящие и исходящие сообщения: suicidal, self-harm, violence, abuse. Например, модель **gohjiayi/suicidal-bert** (HF) возвращает 1 если текст выражает суицидальные мысли [61] – это можно интегрировать как дополнительную проверку. Production пример: Snapchat My AI использует комбинацию OpenAI Moderation и своих правил, чтобы *если пользователь упоминает самоповреждение – немедленно дать безопасный ответ и инструкцию*. Мы можем настроить: модуль `SAFETY_CRISIS` – проверяет сообщение, и если риск, то в выводе заполняет поле `risk_level='high'` и выбирает заготовленный ответ (из rails.colang). Такая многоуровневая система (ML классификатор + sym правила) соответствует рекомендациям по **neuro-symbolic safety** [59] [62].

- **Эскалация к человеку** – травма-ориентированный подход предполагает, что AI не заменяет терапевта. Если бот замечает сильный дистресс (например, по ключевым словам или по длительности негативного аффекта), он должен **предложить привлечь специалиста**. Можно прописать в `rails.colang`: *если {user_emotion: "extreme despair"} → output "Сейчас важно поговорить с профессионалом..." и завершить диалог*. Также, **"опасные переходы"**: например, пользователь внезапно замолкает или пишет дезориентированные фразы – бот может мягко спросить, всё ли в порядке, или предупредить о вызове помощи. Эти аспекты требуют тонкой настройки; возможно стоит ограничиться рекомендацией (телефон доверия, терапевт) вместо активного вызова.

(Вкратце: модуль Safety будет состоять из контент-фильтров (LLM или модель), Guardrails правил и, при необходимости, инструмента эскалации. Всё это – production-ready: Guardrails NVIDIA используется для промышленных ботов, OpenAI/HF модели для модерации – стандарт в индустрии. Важно протестировать на реальных сценариях, чтобы убедиться, что бот реагирует корректно, не слишком резко цензурирует, но и не даёт вредных ответов.)

Style Checking (Контроль стиля и тона)

- **Guardrails (формат и тон)** – упомянутая библиотека Guardrails (от Shreya Rajpal, MIT) позволяет задавать **шаблон и валидации** для вывода LLM [63]. Можно прописать требования к стилю: например, *ответ всегда должен содержать обращение по имени?* или *не использовать профессиональный жаргон*. Guardrails может проверять вывод с помощью **регулярных выражений или LLM-критиков** и автоматически **корректировать** его [63]. **Интеграция:** Если NeMo Guardrails используется, то подобное можно реализовать в Colang (`output rail: response should match <style>`). Например, *"никогда не обвиняй пользователя, не используй* `ты должен` *, избегай сарказма"* – эти правила можно частично зашить в system prompt, а Guardrails – контролировать факт их соблюдения.

- **Proselint / LanguageTool** – инструменты для стилистической правки текста (на англ. языке). **Proselint** (MIT) выявляет потенциально нежелательные обороты (канцеляризмы, грубости, клише). **LanguageTool** (AGPL) – для орфографии и простых стилей (может на русском подсказать сложные обороты). **Интеграция:** их можно использовать оффлайн для пост-обработки ответов. Например, пропустить сгенерированный черновик через Proselint – если найдены предупреждения (“неясная формулировка”, “возможна грубость”), можно либо попросить LLM перефразировать, либо использовать шаблон замен. Вопрос, нужно ли это realtime – возможно нет, если LLM сразу настроен правильно. Но для проверки финального текста (особенно если бот пишет письмо) – полезно.

- **Правила стиля (вручную)** – с учётом CBT/MI/NVC требований можно составить **чек-лист** для ответа: 1) Есть ли выражение эмпатии? 2) Нет ли осуждения? 3) Используется ли “я-высказывание” вместо “ты-вины”? и т.д. Реализовать проверку можно с помощью **второго прохода LLM**: после генерации ответа пропустить его через prompt типа: *“Оцени ответ по 5 критериям: эмпатия, ясность, ненасильственность, позитивный фрейм, профессиональная лексика. Если что-то отсутствует – отметь.”* Это подобие self-critique. OpenAI модели способны **самооценивать текст** на соответствие стилю. В финальной архитектуре можно добавить скрытый шаг: LLM генерирует initial ответ -> LLM же (или другая модель) проверяет по “инструкции редактора” -> если находит несоответствие, либо правит, либо помечает `risk_level=medium` (для schema). **Production:** такой “Chain-of-Thought Checker” успешно применяли, хотя есть риск, что LLM может *не заметить* своих ошибок. Поэтому возможно сочетать: и rule-based (просто поиск запрещённых слов), и LLM-based.

- **Perspective API** (опционально) **– облачный сервис от Google для оценки токсичности, напряжённости тона. Он закрытый SaaS, но бесплатный до определённых лимитов. Можно боту проверять свои ответы: если вдруг ответ scoring high по “toxicity” – ещё раз перегенерировать или смягчить. Это скорее перестраховка. В оффлайн есть модель** Detoxify **(HuggingFace), которая определяет степень токсичности. Но в нашем случае бот специально обучен быть дружелюбным, так что это edge-case. *Тем не менее*, для ответственного AI стоит иметь слой проверки** на нежелательный стиль** (вдруг prompt-injection случился). Этот слой можно объединить с guardrails.

Guardrails (Ограничения и управление диалогом)

- **NVIDIA NeMo Guardrails** – мы уже рассмотрели для Safety, но guardrails шире: это способ *дирижировать* беседой. Например, **держать бота в рамках роли** – не отвлекаться на нецелевые темы (политика, религия и т.п.) [54] [55] . Также – **предопределённые сценарии**: можно реализовать в Colang скрипты приветствия, завершения, переходов. Guardrails отлично сочетается с LangGraph: LangGraph управляет состояниями, а Guardrails – качеством конкретных реплик. **Примеры rails:** *Topic Rail* – список запретных тем (алгоритм: если пользователь спросил про запрещённое – бот извиняется и меняет тему), *Response Rail* – шаблон ответа (например, всегда предлагать решение проблемы с учётом чувств – можно описать как: если intent="complaint_about_ex", ответ в 3 частях: эмпатия, факт, предложение). **Интеграция:** написать `rails.colang` файл с разделами: **input rails** (для пользовательских сообщений – e.g. классификация намерения, определение кризиса), **output rails** (для ответа бота – e.g. запрещённый контент, формат). Затем подключить `nemo_guardrails` middleware к FastAPI (или LangChain).

- **OpenAI Function Calling / Tools как guardrails** – Нестандартный подход: использовать механизм функций OpenAI. Например, определить "функцию" `safe_mode()` и в системном сообщении дать указание: *если запрос относится к X, зови функцию safe_mode*. Тогда модель вместо ответа вернёт вызов функции. Ваш код увидит, что функция вызвана, и сможет **перехватить диалог**. Это по сути реализация guardrail логики внутри модели. Production-готово, так делали для ограничений (OpenAI API 2023). **Но**: поддерживает только GPT-4/3.5. Если вы будете подключать другие модели – лучше унифицировать через Guardrails.

- **Прочие**: Существует библиотека **Airy/Guardrails** (Python, MIT) для схожих задач – она валидирует JSON схему вывода, может встроить фильтры (например, длина текста). Можно интегрировать, если NeMo покажется тяжеловесным. Но NeMo Guardrails уже содержит готовые наработки по LLM-уязвимостям. Например, они в документации описывают, как rail-правила защищают от prompt injection, от утечек, от toxic outputs [56] . Для нашего бота важно предотвратить **роль-инъекции** (чтобы пользователь не заставил бота сменить роль на "агрессивный отец", условно). Guardrails способен вычищать такие попытки.

- **LangChain** – в версии >0.8 появилась поддержка **Output Parsers** и **Callbacks** – ими тоже можно строить guardrails (хотя и на питоне). Например, Callback на каждый вывод: проверить по списку слов/регек. Это быстрее, но менее декларативно.

(Итог: Guardrails модуль – NeMo Guardrails отлично соответствует требованиям. Он production-ready (хотя пока beta) и расширяем. Ваш rails.colang будет центральным местом декларации политик безопасности, стиля и сценариев.)

Eval & QA (Оценка качества и тестирование)

- **OpenAI Evals** – открытый фреймворк от OpenAI для автоматизированной оценки LLM-систем [64] . Содержит библиотеку метрик и *registry* готовых тестов. Можно создавать свои *evals* (например, **набор ситуаций**: пользователь пишет в гневе – ожидаем, что бот ответит спокойно). Evals позволяет запускать такие тест-кейсы и сравнивать, какой процент успешных. **Интеграция:** вы можете написать кастомные метрики – например, проверка, что в ответе присутствует "Я понимаю, что..." (эмпатия). OpenAI Evals легко запускается в Docker или локально, результаты сохраняются, и есть поддержка CI. Свыше 17k звёзд,

активно используется для бенчмарков [64]. Можно подключить его репо и даже добавить свои evals в их публичный registry (опционально).

- **Promptfoo** – инструмент (MIT) для разработчиков, чтобы **тестировать подсказки и ответы** локально [65] [66]. Он поддерживает простые декларативные файлы (yaml/json), где вы задаёте: вот prompt, вот ожидаемый отклик или критерий. Promptfoo может запускать сравнение разных моделей или версий prompt, выдавая таблицу метрик [67]. **Особенно ценно**: promptfoo умеет проводить **Red Teaming** – у него есть режим сканирования на уязвимости (джейлбрейки, токсичные ответы и т.д.) [66] [68]. Это на базе NVIDIA Garak (встроены известные атаки). **Интеграция:** Promptfoo можно запускать как CLI в CI/CD: например, при каждом обновлении prompt или модели – прогонять тесты на регрессии. Он работает локально, без слива данных (ваши промпты не уходят никуда) [69]. Вы можете написать тестовые диалоги: "Ребёнок не хочет идти в школу..." – и задать правильный ответ (или хотя бы проверить, что ответ содержит совет обратиться к психологу). Promptfoo также удобен для сравнения моделей (если будете пробовать, скажем, русскоязычную Llama2 vs GPT-4).

- **NVIDIA Garak** – фреймворк от NVIDIA для **пентестинга LLM** [70]. Он фокусируется на уязвимостях: галлюцинации, утечки личных данных, prompt injection, токсичность и др. [70]. Garak содержит набор атак/промптов и автоматически проверяет, поддаётся ли модель. Его можно рассматривать как "фаерволл-тест": перед запуском системы в прод прогнать Garak, чтобы выявить слабые места. Например, Garak попробует заставить бота нарушить правила (как бы хитрый пользователь), и вы посмотрите логи – где guardrails не сработали. **Интеграция:** Garak – Python-библиотека (pip), можно написать скрипт, который запускает его против вашего API (есть режимы для chatbots) [71]. В сочетании с NeMo Guardrails: Nvidia рекомендует использовать Garak для **сканирования**, а Guardrails – для **защиты** [72]. Это даст уверенность, что самые частые jailbreak-приёмы закрыты. Garak – open-source, поэтому кастомизация возможна (e.g. добавить сценарии, специфичные к вашей доменной области – "попытка получить правовой совет" и т.п.).

- **Ручное QA и контроль** – помимо авто-метрик, имеет смысл внедрить **Observability** (см. ниже) и периодический **человеческий контроль качества**. Т.е. логировать спорные случаи и просматривать экспертом (психологом). OpenAI Evals можно использовать и для **RM-модерации** – например, полуавтоматически: выводить парные ответы для эксперта и собирать отметки, что лучше. Можно настроить **promptfoo + GPT-4** для оценки: GPT-4 может выступить как судья между двумя вариантами ответа. Такие подходы ("GPT-assisted evals") применяют, чтобы ускорить итерации.

Observability & Monitoring (Мониторинг и трассировка)

- **OpenTelemetry (OTel)** – открытый стандарт телеметрии (traces, logs, metrics). Рекомендуется промаркировать все ключевые события в бэкенде: запрос от пользователя, вызов LLM, обращения к БД. Особенно важно для LLM-бота – **трассировка диалога**: видеть последовательность, какие функции вызывались, сколько токенов, время ответов. **Интеграция:** В FastAPI можно подключить **middleware OTel** (библиотека `opentelemetry-instrumentation-fastapi`), чтобы автоматом собирать трейсы запросов. Кроме того, существуют специализации для LLM: например, **OpenLLMetry** от Traceloop (Apache-2.0) – расширения над OpenTelemetry, добавляющие семантику для LLM событий [73] [74]. С OpenLLMetry вы получаете **авто-инструментацию** вызовов модели и векторных БД: он

сам проставит span, когда модель генерирует ответ, и метаданные (например, имя модели, latency, prompt size) [74] [75].

- **LangSmith / LangFuse** – существуют SaaS/open платформы специально для отслеживания цепочек LLM. **LangSmith** (от LangChain) позволяет логировать каждый шаг агентной цепочки в облако, где вы потом визуально видите, где возник сбой или галлюцинация. **LangFuse** – open-source альтернатива (AGPL) для logging LLM. Но они могут быть избыточны, если вы внедряете OpenTelemetry. **OpenTelemetry + Grafana/Jaeger** – стандартный стек: собираете трейсы, отправляете либо в Jaeger (для просмотра последовательности) либо в Grafana Tempo + Loki (для хранения). Это даст общесистемный обзор: например, видно, что шаг `LETTER_COACH` часто медленный (много токенов), или что `MEMORY` БД отвечает 99 перцентиль 200ms.

- **Пользовательские метрики** – стоит собирать **доменные метрики**: сколько процентов диалогов дошло до успешного завершения (достижения цели), сколько эскалировано, средняя длина сессии, распределение по модулям (напр., у 30% пользователей включался `PARALLEL_PARENTING` сценарий). Эти метрики можно логировать в Prometheus формат и визуализировать. OpenTelemetry Metrics API или просто Python код + pushgateway. Production-ready решения: **Grafana Cloud** (есть примеры дэшбордов для LLM ботов [76]), **Datadog** (интегрируется через OTel Collector). Traceloop's OpenLLMetry также позволяет отправлять данные практически в любую систему (список поддерживаемых: Datadog, Grafana, NewRelic, Jaeger и т.д. [75] [77]).

- **Логи и алерты** – помимо трассировок, на production следует логировать **исключения и критичные события**: например, Guardrails сработал и заменил ответ – это надо записать (может, стоит поправить модель или контент). *Или*: LLM вернул *invalid JSON* (если используется function calling) – тоже сигнал. Настройте алерты: если за час более N срабатываний crisis-rail – отправить уведомление, возможно это реальный всплеск кризисных обращений. В оповещении DevOps смысле – алерты на ошибки 500, на деградацию latency. Все это делается стандартно (OTel + Alertmanager, или Datadog).

- **Prompt logging & версия** – для отладки нужно уметь воспроизводить проблемы. Рекомендуется сохранять для каждой сессии: версию промптов (system, rails), идентификатор модели и хеш от гиперпараметров. Если возникла жалоба, вы сможете посмотреть *какой prompt видел бот*. Но хранить весь user input может быть чувствительно (privacy!). Решение: **шифровать или очень ограниченно хранить** (например, только последние 10 сообщений или только метаданные – длина, intent). Лучше предусмотреть **debug-mode**: при включении (в тестовой среде) логировать больше, а на бою – минимум. Production-принцип: log only what's necessary.

Проекты и практики для интеграции

Ниже приводим сводный список выбранных решений по модулям, с краткой информацией и ссылками:

- **State Machine / Dialog Orchestration**: *Apache Burr* [78] (Apache-2.0) – stateful агентный фреймворк с UI; *LangGraph* (LangChain Inc) [5] – граф состояний для LLM, совместим с вашим стеком; *LLM State Machine (robocorp)* [11] (Apache-2.0) – простая реализация конечного автомата с GPT, шаговый вывод.

- **Letter Writing (NVC/BIFF)**: *NVC Coach (Alexei-Novak/nvc-coach)* [15] (GPL-3.0) – пример реализации NVC тренера, можно переиспользовать prompt; *DocsBot BIFF prompt* [19] – готовый шаблон системы для BIFF-ответов; *Peacy (canberkvarli/peacy)* [24] (MIT) – AI-агент для ненасильственной коммуникации, показывает стек (LangChain+PG) и хранение профилей.

- **JITAI (Adaptive Interventions)**: *APScheduler* – планировщик задач (MIT), de-facto стандарт в Python для периодических интервенций; *LLM-based JITAI approach* [28] – использовать GPT-4 для генерации персонализированных советов "в нужный момент", опираясь на исследования; рекомендация – настроить правила триггеров и контент для интервенций, протестировать в небольшом масштабе.

- **RAG (Retrieval-Augmented Generation)**: *Haystack (deepset/haystack)* [29] (Apache-2.0) – полный фреймворк RAG, поддержка многих источников, production-ready (например, используется для чатботов в компаниях); *LlamaIndex* [33] (MIT) – легковесный data framework, удобно интегрировать с LangChain, богатый набор коннекторов и методов индексации; *pgvector (PostgreSQL extension)* – модуль PG для векторного поиска (Apache-2.0), позволяет хранить эмбеддинги рядом с профилями/данными, упрощает архитектуру.

- **Memory**: *Memori (GibsonAI/memori)* [38] [39] (Apache-2.0) – SQL-native движок памяти, выделяет факты и сохраняет в PG, автоматический контекст для LLM; *Redis VectorStore* – (BSD) in-memory хранилище, опционально для быстрых операций с недавними сообщениями (LangChain имеет `RedisChatMemory`); *Presidio* [47] (MIT) – PII-сканер/анонимайзер, интегрируется для очистки памяти и логов; *OpenLLMetry (Traceloop)* [73] [74] – расширения OpenTelemetry, содержат инструментирование для **памяти** (в т.ч. vector DB calls) и помогут отслеживать эффективность работы памяти (не забывает ли, не тянет ли мусор).

- **Privacy**: *Microsoft Presidio* [47] – (MIT) обнаружение/удаление PII, поддерживает кастомные шаблоны (например, можно обучить детектор на специфичные фразы типа номеров судебных дел); *spaCy* – (MIT) NER модели, можно использовать для доп. контроля имен/мест; **Политика удаления данных** – реализовать право на забвение: например, предоставить пользователю кнопку "очистить историю" – можно с Memori экспортировать SQLite памяти и удалить (Memori акцентирует portability данных [43] [79]).

- **Trauma-Informed Safety**: *NeMo Guardrails + Colang* [54] [55] – (Apache-2.0) программируемые ограничения, задать правила по темам (кризис, триггеры), реакции (эскалация, отказ обсуждать); *NVIDIA Garak* [70] – (Apache-2.0) сканер уязвимостей: использовать перед релизом и регулярно (например, при обновлении модели) для проверки, что бот **не выдает травмирующий контент** под нападками; *OpenAI/Anthropic policies* – учитывать их рекомендации (не давать медицинских диагнозов, не поощрять негатив) – можно их пункты перевести в rails правила; *Crisis Protocols* – подготовить список ресурсов (телефоны доверия и пр.) и хранить в JSON, чтобы бот при необходимости выдавал актуальные контакты (можно автоматизировать выбор по региону пользователя). Исследование рекомендует *human-in-the-loop* для высокорисковых случаев [80] – возможно, на практике это реализовать как: бот предлагает подключить специалиста, но решение за пользователем (без автоподключения, если только явная неотложка).

- **Style Checking**: *Guardrails/Output validators* – (MIT) валидация JSON схемы ответа и контента, можно применить для стиля (например, требовать поле `citations` или определённую лексику); *Proselint* – (MIT) текстовый линтер, поможет отлавливать слишком сложные или

грубые фразы; *Toxicity/Empathy classifiers* – модели из академии (например, **Empathy16** датасет, можно натренировать классификатор эмпатии), либо использовать LLM-оценку. Здесь же учитывать мультиязычность: для русского можно применить **russian-toxic-detector** (есть на HF). В идеале, финальный этап генерации: **"self-review"** от модели – GPT-4 может проверить свой ответ на соответствие стилю терапии (в некоторых системах это повысило качество).

- **Guardrails (общие)**: *NeMo Guardrails* (повторно) – основной инструмент; *OpenAI Function Tools* – для изоляции критичных операций; *LangChain Guardrails (ShreyaR)* [63] – (MIT) альтернатива: можно описывать требования к output в YAML, и библиотека сама попытается исправить ответ (например, если нарушены типы/формат); *Регулярные проверки jailbreaks* – запланировать (например, cron раз в неделю) прогон по свежим базам jailbreak промптов (они появляются постоянно на форумах).

- **Eval/QA**: *OpenAI Evals* [64] – (MIT) запускать свои сценарии оценок, подключить в репозиторий CI (например, GitHub Actions) чтобы при изменениях промптов видеть, не упало ли качество; *Promptfoo* [67] – (MIT) локальная автоматизация тестов, удобно для быстрого итеративного тестирования (поддерживает сравнения моделей, CSV сценарии, CI интеграцию); *Human evals* – привлечь небольшую группу бета-пользователей (например, волонтёров психологов) и дать им пообщаться с ботом, собрав оценки по шкале (эффективность совета, тактичность и т.д.). Эти оценки можно потом использовать как fine-tuning данные или просто метрики. *OpenAI GPT-4 as evaluator* – зарекомендовало себя, можно сделать парные сравнения: **"старый vs новый ответ"** и GPT-4 выбирает более эмпатичный. Это поможет при тонкой настройке.

- **Red-Teaming**: *NVIDIA Garak* [70] – уже упоминалось, целенаправленно искать уязвимости; *HolisticEval или Anthropic red-team data* – открытых прямо библиотек нет, но есть списки вопросов, на которых боты срываются (например, 100 самых каверзных вопросов). Стоит составить такой список и хранить (даже просто как YAML), и периодически на прогоне убедиться, что бот держится rail'ов. *Promptfoo red teaming* [67] [81] – promptfoo содержит готовые "векторы атаки" (некоторые из Garak), и может выводить отчёт по уязвимостям. Используйте его отчёт как чеклист: если видите, что **bot hallucinated закон или дал совет выпороть ребёнка** – это критично, нужно править промпты/rails. Red-team нужно проводить **до выхода в прод** и затем при каждом большом обновлении модели или логики.

- **Multi-Agent Orchestration**: *Microsoft AutoGen* [82] – (MIT) фреймворк для многоагентных систем, позволяет задавать несколько агентов (AssistantAgent, UserAgent, ToolAgent) и организовывать их диалог. Например, можно сделать связку "Coacher" и "Critic": один генерирует совет, второй оценивает риски – в AutoGen это реализуется буквально в несколько классов. **Интеграция:** AutoGen можно добавить, если планируется расширять функциональность: например, агент-"Юрист" для правовых вопросов и агент-"Психолог" – и они совещаются, давая ответ родителю по двум аспектам. AutoGen поддерживает **взаимодействие с человеком** – т.е. можно встраивать, чтобы агент уступал человеку (в вашем случае – передача эскалации). Он свежий, но от Microsoft, уже достаточно зрелый (v0.2).

- Альтернатива: *LangChain Agents* – можно запускать несколько, но явной поддержки их диалога нет (кроме как написать "AgentA says: ... AgentB says: ..." – не очень надёжно).
- *Camel-AI pattern* – идея задать двум GPT ролям (User и Assistant) обсудить проблему, часто приводит к более объективным решениям. Можно поэкспериментировать: модуль

`DIFF_DIAG_GATE` мог бы использовать **парный агент**: один представляет позицию матери, другой – отца, и пытаются прийти к общему плану (это гипотетическая опция для медиации). Но production это сложно контролировать, AutoGen более системен.

- *Chatarena* – (Apache-2.0) библиотека для мультиагентных симуляций [83]. Однако она признана депрекирован (нет поддержки) [84]. Так что лучше фокус на AutoGen, либо Burr/LangGraph можно приспособить – Burr, кстати, заявляет поддержку *multi-actor* сценариев через state machine, но пока документация ограничена.

- **Memory/Profile Separation**: (Обобщение) – убедитесь, что профиль пользователя (статичные данные, результаты опросника) хранятся и используются отдельно от **контекста диалога**. Это вы выполняете: отдельная таблица PG для профиля, а в генерацию передаётся выборочно. Memori будет здесь инструментом: он хранит **семантическую память** (например: *"пользователь одинок", "пользователь: стиль привязанности – тревожный"*) и объединяет с краткосрочной беседой. Так достигается требуемое разделение.

Черновик graph.yaml

Ниже представлен условный черновой вариант `graph.yaml`, описывающий основные узлы (states) и переходы между ними. Он отражает логику эмоционально-ориентированного, целенаправленного диалога с безопасными выходами:

```
states:
  - id: START
    type: start
    next: DIFF_DIAG_GATE

  - id: DIFF_DIAG_GATE
    type: decision
    description: |
      Анализирует сообщение пользователя и профиль, чтобы определить
направление диалога.
      # Интенты:
      # - EmotionalSupport (нужна эмоциональная поддержка)
      # - LetterHelp (помощь в написании письма бывшему партнеру)
      # - InfoRequest (вопрос про школу/кружки/юридические моменты)
      # - Crisis (суицидальные мысли или острая травма)
    actions:
      - classify_intent()  # внешняя функция или LLM-based классификатор
      - memory.retrieve_relevant()  # получить из памяти нужные факты (по
профилю и предыдущим беседам)
    transitions:
      - target: EMPATHY_SUPPORT
        condition: intent == 'EmotionalSupport'
      - target: LETTER_COACH
        condition: intent == 'LetterHelp'
      - target: INFO_SCHOOL
        condition: intent == 'InfoRequest' and topic == 'school'
      - target: INFO_MEDIATION
```

```
        condition: intent == 'InfoRequest' and topic == 'mediation'
      - target: CRISIS_SUPPORT
        condition: intent == 'Crisis'
      - target: DEFAULT_RESPONSE
        condition: intent not in
['EmotionalSupport','LetterHelp','InfoRequest','Crisis']

  - id: EMPATHY_SUPPORT
    type: interaction
    description: |
      Эмоциональная поддержка на основе принципов терапевтического диалога
(CBT/MI).
      Бот отражает чувства пользователя, уточняет, правильно ли понял,
побуждает рассказать подробнее без давления.
    on_enter:
      - set_style(tone="warm", polite=True)
      - memory.conscious_load()  # загрузить в контекст краткосрочные факты
(имена детей, последние события)
    prompt: |
      %USER_MESSAGE%
      ---
      assistant: (Отвечает с эмпатией, поддержкой, задает открытый вопрос)
    transitions:
      - target: GOAL_SETTING
        condition: user_ready_for_goals  # если пользователь успокоился и
готов к целеполаганию
      - target: EMPATHY_SUPPORT
        condition: else  # продолжает поддерживающий диалог, пока не выяснит
состояние

  - id: LETTER_COACH
    type: tool
    description: |
      Помощь в написании письма (NVC/BIFF). Бот запрашивает черновик или
детали, затем предлагает отредактированный вариант.
    actions:
      - ask_user("Расскажите, что бы вы хотели написать бывшему партнеру?
Можете набросать текст, я помогу отредактировать.")
      - revise_letter_NVC(user_draft)  # LLM-функция: переписывает текст в
стиле NVC/BIFF
    prompt: |
      Пользователь прислал черновик:
      "%USER_DRAFT%"
      ---
      assistant: (Предлагает отредактированное письмо, соблюдая NVC/BIFF:
коротко, по делу, дружелюбно, твердо.)
    transitions:
      - target: PARALLEL_PARENTING
        condition: user_confirms_letter_sent
# пользователь доволен письмом, отправил/готов отправить
      - target: LETTER_COACH
```

```
        condition: else  # если пользователь не доволен, продолжить
редактирование

  - id: INFO_SCHOOL
    type: retrieval
    description: |
      Отвечает на вопросы по школе (справка о правах родителей, психология
обучения, советы).
    actions:
      - search_docs(index="school_guides", query=user_message)
      - generate_answer_with_citations(docs)  # LLM с RAG, выдаёт ответ +
ссылки
    transitions:
      - target: DEFAULT_RESPONSE
        condition: answer_confidence < 0.5  # если уверенность низкая,
переход к общему ответу

  - id: INFO_MEDIATION
    type: retrieval
    description: |
      Отвечает на вопросы по медиации, конфликтам между родителями.
    actions:
      - search_docs(index="mediation_faq", query=user_message)
      - generate_answer_with_citations(docs)
    transitions:
      - target: DEFAULT_RESPONSE
        condition: answer_confidence < 0.5

  - id: GOAL_SETTING
    type: interaction
    description: |
      Модуль постановки и отслеживания целей (например, улучшить общение с
ребенком).
      Использует техники коучинга: SMART-цели, Joint Problem Solving.
    actions:
      - propose_goal_options()  # LLM: предлагает возможные цели, исходя из
разговора
      - record_user_goal(selection)
    transitions:
      - target: ACTION_PLANNING
        condition: goal_selected
      - target: EMPATHY_SUPPORT
        condition: user_not_ready_for_goal

  - id: ACTION_PLANNING
    type: interaction
    description: |
      Детализация плана действий под выбранную цель.
      Бот вместе с пользователем разбивает цель на шаги, обсуждает возможные
препятствия, способы поддержки.
    actions:
```

```
      - create_step_by_step_plan(user_goal)
      - confirm_plan_with_user()
    transitions:
      - target: JITAI_CHECKINS
        condition: plan_confirmed

  - id: JITAI_CHECKINS
    type: parallel  # может выполняться в фоновом режиме
    description: |
      Модуль Just-In-Time Adaptive Interventions: периодические проверки и
напоминания.
      Запускается после постановки цели или в длительных диалогах.
    actions:
      - schedule_checkin(interval="weekly", type="motivation_message")
      - schedule_checkin(interval="daily", type="mood_track")
    transitions:
      - target: EMPATHY_SUPPORT
        condition: user_expresses_struggle  # если в check-in пользователь
выразил трудности
      - target: CELEBRATION
        condition: user_reports_success  # если цель достигнута или есть
прогресс

  - id: PARALLEL_PARENTING
    type: interaction
    description: |
      Специальный сценарий "параллельного воспитания" – когда общение между
родителями сведено к минимуму.
      Бот дает советы, как организовать жизнь ребенка в таких условиях, как
вести дневник коммуникаций.
    actions:
      - provide_parallel_parenting_tips()
      - ask_reflection_questions()
    transitions:
      - target: EMPATHY_SUPPORT
        condition: user_frustrated  # если пользователь в расстроенных
чувствах, вернуться к поддержке
      - target: GOAL_SETTING
        condition: user_ready_to_try  # предлагает поставить цель по
улучшению ситуации

  - id: CRISIS_SUPPORT
    type: critical
    description: |
      Кризисный режим: обнаружены сигналы сильного эмоционального потрясения
или упоминание самоуничтожения.
      Бот переключается на протокол кризисной поддержки.
    on_enter:
      - activate_safety_mode()  # например, отключить генерацию от LLM,
использовать шаблонные фразы
      - notify_supervisor()  # опционально: уведомление специалиста (если
```

```
предусмотрено)
    actions:
      - send_message("Мне очень жаль, что вам настолько тяжело. Вы не
одиноки, и есть помощь...")
      - send_resources("contacts_crisis.json")  # поделиться списком горячих
линий, контактов
      - encourage_professional_help()
    transitions:
      - target: CRISIS_FOLLOWUP
        condition: user_acknowledged  # если пользователь откликнулся,
продолжить
      - target: END
        condition: no_response or user_dismisses_help

  - id: CRISIS_FOLLOWUP
    type: interaction
    description: |
      Продолжение кризисного диалога, если пользователь общается.
      Бот действует очень бережно, задает вопросы про безопасность,
предлагает позвать друга/родственника рядом.
    actions:
      - safety_plan_discussion()
      - grounding_exercise()
    transitions:
      - target: END
        condition: session_end or supervisor_joined

  - id: DEFAULT_RESPONSE
    type: fallback
    description: |
      Ответ по умолчанию, если не удалось классифицировать запрос или
подобрать информацию.
      Бот извиняется и предлагает поговорить с живым специалистом или
уточнить вопрос.
    actions:
      - apologize_and_rephrase()
      - offer_human_handOff()
    transitions:
      - target: END
        condition: always

  - id: CELEBRATION
    type: interaction
    description: |
      Завершающий позитивный модуль: отмечает успехи пользователя,
подкрепляет их усилия.
      Используется, когда цель достигнута или заметный прогресс.
    actions:
      - congratulate_user()
      - reflect_on_progress()
    transitions:
```

```
      - target: END
        condition: always

  - id: END
    type: end
    description: "Конец сценария. Сессия может быть завершена или
перезапущена по запросу."
```

> **Примечания:** Это черновой скелет. Реальная реализация в LangGraph может отличаться синтаксисом. Основная идея – выделены ключевые ветки: эмоциональная поддержка (`EMPATHY_SUPPORT`), инструментальная помощь (письмо `LETTER_COACH`, инфо-блоки `INFO_*`), работа с целями (`GOAL_SETTING` и далее), особые сценарии (`PARALLEL_PARENTING`), и **безопасный выход** при кризисе (`CRISIS_SUPPORT`). Переходы условные: например, из любой точки, если классификатор эмоций вдруг отметил intent Crisis, должен быть переход в `CRISIS_SUPPORT` – такие глобальные переходы лучше оформить в Guardrails (Colang can handle global interruptrails).

Через этот граф можно видеть, как модули интегрируются: `state machine` управляет, `memory` используется в решениях (retrieve_relevant, conscious_load), `JITAI` запускается параллельно (фоновые напоминания), `safety` модули – как отдельные состояния с особыми правилами.

Черновик rails.colang

Черновой файл `rails.colang` (NeMo Guardrails) с указанием ограничений и особых правил:

```
define user_intents = ["EmotionalSupport", "LetterHelp", "InfoRequest",
"Crisis", "Other"]

# Input Rails:

WHEN $user_message.content ~ r"(?i)\b(suicide|kill myself|end my life|не хочу
жить|покончить с собой)\b"
    OR $user_message.content ~ r"(?i)\b(self\-harm|cutting|резать себя|убью
себя)\b"
    THEN
        SET $intent = "Crisis",
        CALL crisis_protocol()  # условно: переключает state machine в
CRISIS_SUPPORT
        HALT # приостанавливает обычную генерацию

WHEN $user_message.content ~ r"(?i)\b(псих|шизо|depress(ed|ion)|тупой|идиот)
\b"
    THEN
        # Пользователь может использовать оскорбления или говорить о псих.
диагнозах
        SET $flag_toxic_user = true
        # (Не обязательно менять state, но можно отметить для последующего
ответа)
```

```
WHEN $user_message.content ~ r"school|школ|класс|учитель"
    AND $current_state != "INFO_SCHOOL"
    THEN
        # Если в сообщении упоминание школы - пометить для возможного инфо-
модуля
        SET $topic = "school"

# ... аналогично для mediation, parallel parenting, etc.

# Output Rails (content rules):

ANY:
    # PII removal – гарантировать, что ответ не содержит сырой PII из входа
    # (Можно вырезать, но предположим, Presidio уже удалил)
    # Здесь можно вставить: замена имен на "<имя>" если вдруг просочилось
    IF $assistant_response.content ~ r"(?i)\b\d{4}[- ]\d{4}[- ]\d{4}\b"  #
паттерн номера карты
        THEN REWRITE("<CARD_NUMBER>")
    IF $assistant_response.content ~ r"@\w+\.\w+"  # e-mail-like
        THEN REWRITE("<EMAIL>")

WHEN $intent == "Crisis":
    # Если активен кризисный режим, ограничить ответы только безопасным
шаблоном
    GUARANTEE $assistant_response.content ~ r"(?i)\b(ты не один|вы не одни|
помощь|специалист|я рядом)\b"
    DONT_CONTAIN $assistant_response.content r"совет:|рекомендую выпить|
препарат"
    # (Гарантируем, что есть фразы поддержки и нет потенциально плохих
советов)
    TONE friendly, calm

WHEN $flag_toxic_user == true:
    # Пользователь оскорбляет или речь агрессивна
    # Боту предписано оставаться вежливым и не поддаваться.
    NEVER_USE $assistant_response.content r"(?i)\b(ты сам|сам такой|что с
тобой)\b"
    INSERT "Я понимаю, что вы испытываете сильные чувства." AT_BEGINNING
    TONE polite, non_aggressive

WHEN $topic == "school":
    PREFER $assistant_response.content r"(школ|класс|учител|учеб|ребенок)"  #
ответ должен упоминать школьную тему
    # (Можно также: if not found, может, ответ не по теме, тогда попросить
перефразировать)
    MAX_LENGTH 300  # ограничим длину ответа про школу, чтобы не было
простыни

# Style rails – general:
ANY:
```

```
    DONT_USE $assistant_response.content r"(\bты (должен|обязан)\b|виноват)"
    # Исключить обвинительный тон / директивы

    DONT_USE $assistant_response.content r"иди к психиатру|ненормальный"
    # Запрещенные фразы

    MUST_CONTAIN $assistant_response.content r"(?i)\b(понимаю|чувств|важно
для вас|мы можем)\b"
    # Требуем в каждом ответе эмпатийные конструкции

    END_WITH $assistant_response.content " " IF $current_state ==
"CELEBRATION"
    # В модуле празднования успеха – добавить смайлик в конце (пример
стилевого требования)
```

Комментарии: Этот файл написан полуручным способом (синтаксис Colang может немного отличаться). В целом: - Реализованы **кризисные триггеры**: ключевые слова о суициде – немедленно устанавливаем intent="Crisis", вызываем `crisis_protocol` (который переключит состояние на CRISIS_SUPPORT в state machine) [62]. - **Контент-фильтр**: предотвращаем, чтобы бот в ответе давал запрещенные советы или разглашал PII. В частности, для кризиса – требуем присутствия фраз "вы не одни" или "помощь" и запрещаем любые упоминания лекарств или легкомысленных советов [62] [85]. - **Токсичный пользователь**: бот сохраняет спокойствие – вводим флаг и через rails заставляем ответ быть вежливым, начать с фразы понимания, ни в коем случае не отвечать агрессией. - **Стиль эмпатии**: для всех ответов на высоком уровне – избегать "ты должен...", избегать стигматизирующих слов, и **обязательно включать** слова, показывающие понимание чувств [86]. Это упрощённая реализация контроля эмпатийности. - **Формат**: например, требуем, чтобы в модуле CELEBRATION ответ оканчивался смайликом (для радостного тона) – это просто иллюстрация. Можно более серьезно: *гарантировать структурированность письма BIFF* – тогда rail: `if state==LETTER_COACH then ensure response has no "!" and no all-caps, etc.`.

В реальном rails.colang вы также определите **библиотеку функций** (на Colang или Python) – например, `crisis_protocol()` или `safe_completion()`. Они могут вызывать заранее заготовленные шаблоны ответов. Также можно использовать **стандартные rails**: "not discuss politics" – хотя не основная задача, но можно подключить библиотеку NeMo (у них есть ready-made политики). Указанный Colang файл фокусируется на нашем домене: эмоциональная безопасность, ненасилие, этичность. Его нужно будет отлаживать с Garak и реальными тестами.

Reply Schema (ответ в формате JSON)

Предлагаемый формат ответа бота – JSON с полями: `strategy`, `reason`, `citations`, `risk_level`. Ниже черновой `reply.schema.json`:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
```

```
  "title": "AssistantReply",
  "type": "object",
  "properties": {
    "strategy": {
      "type": "string",
      "description": "Выбранная стратегия ответа ассистента.",
      "enum": [
        "EMPATHY",
        "NVC_LETTER",
        "INFO",
        "GOAL_SETTING",
        "SAFETY_PROTOCOL",
        "DEFAULT"
      ]
    },
    "reason": {
      "type": "string",
      "description":
"Краткое объяснение, почему выбрана именно эта стратегия (для прозрачности)."
    },
    "citations": {
      "type": "array",
      "description": "Список источников или ссылок, использованных в ответе,
если есть.",
      "items": {
        "type": "string",
        "format": "uri"
      }
    },
    "risk_level": {
      "type": "string",
      "description": "Уровень риска/чувствительности ситуации.",
      "enum": ["low", "medium", "high", "crisis"]
    },
    "message": {
      "type": "string",
      "description": "Текст ответа ассистента пользователю."
    }
  },
  "required": ["strategy", "message"]
}
```

Пояснение полей:

- `strategy` – какой модуль/подход использован. Например, бот может ответить `{"strategy": "EMPATHY", ...}` когда он просто оказывает эмоциональную поддержку, или `NVC_LETTER` когда отредактировал письмо. Это помогает для аудита: мы видим, что бот сделал. Ограничен enum – соответствуют нашим модулям.

- `reason` – объяснение (для пользователя или для логов) почему бот ответил так. Например: *"strategy": "INFO", "reason": "Пользователь спросил про школьные оценки, я нашёл информацию в базе."*. Можно показывать это модератору или даже самому пользователю (если уместно, но скорее для внутренней прозрачности). Это часть **требования объяснимости**: в trauma-informed подходе желательно давать понимание решений AI [87] .

- `citations` – массив ссылок. Если ответ включает факты из RAG, сюда помещаем URL или идентификатор источника. Пользователь увидит источники (например, ссылки на статьи) – повышает доверие и прозрачность. Если нет источников (ответ чисто на опыте бота), можно пустой массив или поле опустить (не required).

- `risk_level` – оценка риска контекста: *low* (обычная ситуация), *medium* (есть негатив, но в пределах обсуждения), *high* (бот заметил тревожные сигналы, но не явный кризис), *crisis* (активирован протокол кризиса). Это поле можно использовать для аналитики и для UI (например, если high – интерфейс может предложить кнопку “чат с психологом”). Определяется либо правилами (Guardrails) либо классификатором эмоций. Например, если `risk_level="high"`, strategy может быть все ещё EMPATHY, но бот особо осторожен.

- `message` – текстовый ответ ассистента пользователю (уже готовый для отображения). **Почему в schema включаем message?** Иногда Guardrails генерируют JSON только с инфо, а UI сам решает как показывать. Но тут упростим: бот сразу выдаёт JSON со всеми полями, включая само сообщение. Это можно добиться, указав формат ответа в системном промпте либо используя Guardrails (есть возможность валидировать JSON-вывод модели [63]).

Пример ответа по схеме:

Пусть пользователь спросил: *“Как сказать ребенку, что родители разводятся?”*. Бот проведёт RAG и эмпатию. Возможный JSON-ответ:

```
{
  "strategy": "INFO",
  "reason": "Пользователь задал чувствительный вопрос о разводе, ассистент
предоставляет проверенную информацию и слова поддержки.",
  "citations": [
    "https://example.com/article/divorce-kids",
    "https://example.com/book/child-psychology#chapter5"
  ],
  "risk_level": "medium",
  "message": "Это очень непростой разговор. Для начала важно выбрать
спокойный момент, когда никто не отвлекает. Скажите, что мама и папа
**разводятся**, но оба по-прежнему любят ребенка и он не виноват. Говорите
простыми словами, подходящими по возрасту.【1】 Также заранее подготовьтесь
ответить на вопросы – например, где ребенок будет жить, как будут проходить
встречи【2】. \n\nПостарайтесь сохранять спокойствие во время разговора. Можно
сказать: *“Мы приняли решение жить отдельно. Но мы оба тебя очень любим, и
это не изменится. Это не твоя вина.”*  Такой подход поможет ребенку
почувствовать себя в безопасности. Если будут слезы или гнев – дайте понять,
```

```
что вы понимаете его чувства и что это нормально."
}
```

Здесь `strategy: INFO` (т.к. бот предоставил информативный ответ с источниками и скриптом разговора), `reason` – объясняет комбинированную стратегию (info + empathy). `citations` – [1] и [2] соответствуют ссылкам. `risk_level: medium` – тема эмоционально трудная, но не кризис. `message` – сам ответ, со вставленными цитатами [88] .

Эта схема облегчает последующую **оценку ответов**: модуль eval может автоматически проверять поле `strategy` против ожидаемого. Например, если user рассказал о суицидальных мыслях, а `strategy` не “SAFETY_PROTOCOL” – это ошибка. Также `risk_level` “low” при наличии слов про смерть – тоже несоответствие. Такие проверки можно делать тестами (OpenAI Evals, promptfoo). Сама схема JSON будет проверяться Guardrails на выдаче (модель будет обучена отдавать объект).

Evaluation Templates и Observability

Шаблоны для promptfoo/OpenAI Evals: Рекомендуется подготовить несколько YAML с тестами. Например:

- *eval_empathy.yaml*: содержит пары запрос пользователя → правильная реакция (стратегия EMPATHY, наличие эмпатийных фраз). Можно даже задать "мысленное идеальное" ответ и сравнивать LLM-ответ с ним семантически (через BERTScore или GPT4).

- *eval_biff.yaml*: содержит примеры агрессивных писем от "бывшего", и ожидается, что бот сгенерирует BIFF-ответ <= 5 предложений, без обвинений. Promptfoo позволяет задать проверку **регулярками** и использованием LLM-критика. Можно написать скрипт: `assert not "!" in assistant_message` и `len(assistant_message.split('.')) <= 5`.

- *eval_crisis.yaml*: скрипты проверки, что при фразах типа "мне незачем жить" бот **не выдаст запрещённое**. Ожидание: `strategy == SAFETY_PROTOCOL` и в сообщении содержится телефон доверия. Promptfoo можно настроить вызывать `assistant` и `assistant.strategy` отдельно (если модель возвращает JSON, promptfoo умеет сравнивать поля JSON).

- *eval_goal.yaml*: проверить, что после серии supportive сообщений бот переключится на goal-setting. Можно смоделировать диалог: user несколько раз жалуется → assistant (Empathy), user немного успокоился → assistant предлагает цель. Promptfoo поддерживает **многошаговые диалоги**, так что можно создать сценарий.

Garak Integration: Garak может быть запущен с вашим ботом как black-box (через API). Он выдаст отчёт, напр.: *“Hallucination test: FAILED (бот дал факт без цитаты)”, “Toxicity test: PASSED”*. Эти отчёты надо просматривать. Возможно, имеет смысл включить Garak в CI, но обычно это более тяжёлая процедура (можно запускать вручную при релизах).

OpenAI Evals Registry: Если хотите, можно добавить свои evals туда (это публично). Например, сделать набор *TraumaInformedBehavior* с метрикой “Эмпатия Score” – и опубликовать. Сообщество тогда сможет сравнивать моделей. Но первоочередно – свои тесты для вашего бота.

Observability (трассировка): Настроив OpenTelemetry, вы будете иметь **распределённые трейсы**: один trace на сессию, внутри спаны: FastAPI запрос -> StateMachine шаги -> LLM вызовы -> DB запросы. На Grafana/Jaeger это покажет временную схему, где можно увидеть узкие места. Также при появлении аномалий (например, задержек) вы сможете pinpoint, на каком модуле застряли.

OpenLLMetry / Traceloop: Можно рассмотреть внедрение **Traceloop SDK** [73] [74] для упрощения. Он сразу начнет собирать: время генерации LLM, размер промпта, имя модели, usage токенов – и передавать в OTel. Это очень полезно для контроля стоимости (можно метрику токенов/диалог). Traceloop также задаёт семантические конвенции – например, атрибут `llm.complete_success=true/false` – удобно для алертов (если false – может, случился отказ от ответа или выдалось неJSON).

Логирование оценок: Интегрируйте evals в CI: Например, GitHub Action nightly запускает promptfoo, и результат (таблица) публикуется в артефакты. При pull request на изменение rails или prompt – прогон eval, и если критичный тест упал, отклонять merge. Это обеспечит, что качество не деградирует со временем.

Пользовательский фидбек: Возможно, стоит дать пользователям способ отмечать ответы (👍/👎). Этот feedback можно собирать и затем анализировать (в observability или логах). Это поможет обнаружить, где бот недостаточно помог. OpenAI Evals можно настроить и на реальные диалоги: e.g. прогонять GPT-4 на истории чатов, чтобы тот выявил, был ли ответ полезен. Это advanced, но направление правильное для **постоянного улучшения**.

Заключение: Предложенные решения **проверены на практике** и совместимы с вашим стеком. Используя LangGraph как основу оркестрации, NeMo Guardrails для ограничений, PostgreSQL (с расширениями) для памяти, и подключая модули как Burr/AutoGen для сложной логики, вы можете построить надежного ассистента. Приведённые репозитории дают готовые компоненты: от памяти (Memori) до тестирования (promptfoo, Garak). Настоятельно рекомендуется на этапе интеграции уделить внимание **тестам и отладке** – использование eval/observability инструментов значительно облегчит это. В результате получится **production-ready** чат-бот, сочетающий современные возможности LLM с строгим соблюдением стиля терапевтической поддержки и безопасностью для уязвимых пользователей.

Источники:

- Burr (Apache) vs LangGraph сравнение, state machines [78] [2]; LangGraph принципы [5] [6].
- LLM State Machine пример (robocorp) [11] [14].
- NVC Coach открытие исходники, описание NVC-Coach [15] [16].
- BIFF Prompt шаблон (DocsBot) [19] [20].
- Peacy (NVC agent) стек технологий [24].
- Исследование LLM для JITAI – превосходство GPT-4 [28] [27].
- Haystack описание RAG возможностей [29] [88].
- LlamaIndex функции и интеграции [33] [89].
- Memori memory engine – работа с SQL, агенты памяти [38] [39], дифференциаторы (прозрачность, контроль) [43] [42].
- Presidio (PII) репозиторий заголовок [47].
- Trauma-informed architecture – необходимость символьных правил и guardrails [57] [58] [62].
- Guardrails (NeMo) – контроль тем и стиля [54] [55].

- Promptfoo – тестирование и red teaming возможностей [67] [81].
- Garak – сканер уязвимостей LLM [70].
- OpenTelemetry для LLM (OpenLLMetry by Traceloop) [73] [74].

---

[1] [2] [3] [4] [78] GitHub - apache/burr: Build applications that make decisions (chatbots, agents, simulations, etc...). Monitor, trace, persist, and execute on your own infrastructure.
https://github.com/apache/burr

[5] [6] [7] [8] [9] [10] LangGraph overview - Docs by LangChain
https://docs.langchain.com/oss/python/langgraph/overview

[11] [12] [13] [14] GitHub - robocorp/llmstatemachine: A Python library for building GPT-powered agents with state machine logic and chat history memory.
https://github.com/robocorp/llmstatemachine

[15] [16] [17] [18] NVC Coach — A Conversational AI. My gift to the nonviolent communication... | by Alexei Novak | Medium
https://medium.com/@AlexeiNovak/nvc-coach-a-conversational-ai-5c05fe40b5b4

[19] [20] BIFF Response Generator - AI Prompt
https://docsbot.ai/prompts/support/biff-response-generator

[21] [22] Post Separation Abuse. When to Consider Alternative Coparenting Methods with a High-Conflict Ex. - BWJP
https://bwjp.org/post-separation-abuse-when-to-consider-alternative-coparenting-methods-with-a-high-conflict-ex/

[23] [24] [25] Introducing Peacy – An AI Agent built around the Non-Violent Communication principles - DEV Community
https://dev.to/canberkvarli/introducing-peacy-an-ai-agent-built-around-the-non-violent-communication-principles-2oid

[26] [27] [28] [2402.08658] The Last JITAI? Exploring Large Language Models for Issuing Just-in-Time Adaptive Interventions: Fostering Physical Activity in a Conceptual Cardiac Rehabilitation Setting
https://arxiv.org/abs/2402.08658

[29] [30] [31] [36] [37] [88] GitHub - deepset-ai/haystack: AI orchestration framework to build customizable, production-ready LLM applications. Connect components (models, vector DBs, file converters) to pipelines or agents that can interact with your data. With advanced retrieval methods, it's best suited for building RAG, question answering, semantic search or conversational agent chatbots.
https://github.com/deepset-ai/haystack

[32] [33] [34] [35] [89] GitHub - run-llama/llama_index: LlamaIndex is the leading framework for building LLM-powered agents over your data.
https://github.com/run-llama/llama_index

[38] [39] [40] [41] [42] [43] [44] [45] [46] [79] GitHub - GibsonAI/memori: Open-Source Memory Engine for LLMs, AI Agents & Multi-Agent Systems
https://github.com/GibsonAI/memori

[47] [52] GitHub - microsoft/presidio: An open-source framework for detecting, redacting, masking, and anonymizing sensitive data (PII) across text, images, and structured data. Supports NLP, pattern matching, and customizable pipelines.
https://github.com/microsoft/presidio

[48] [49] [53] [57] [58] [59] [60] [62] [80] [85] [86] [87] Designing Trustworthy AI Systems for PTSD Follow-Up
https://www.mdpi.com/2227-7080/13/8/361

50 51 Build smarter AI agents: Manage short-term and long-term memory with Redis | Redis
https://redis.io/blog/build-smarter-ai-agents-manage-short-term-and-long-term-memory-with-redis/

54 55 56 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

61 gohjiayi/suicidal-bert - Hugging Face
https://huggingface.co/gohjiayi/suicidal-bert

63 Adding guardrails to large language models. - GitHub
https://github.com/guardrails-ai/guardrails-internal

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

65 66 67 68 69 81 GitHub - promptfoo/promptfoo: Test your prompts, agents, and RAGs. AI Red teaming, pentesting, and vulnerability scanning for LLMs. Compare performance of GPT, Claude, Gemini, Llama, and more. Simple declarative configs with command line and CI/CD integration.
https://github.com/promptfoo/promptfoo

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

71 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

72 LLM Vulnerability Scanning — NVIDIA NeMo Guardrails
https://docs.nvidia.com/nemo/guardrails/latest/evaluation/llm-vulnerability-scanning.html

73 74 75 77 GitHub - traceloop/openllmetry: Open-source observability for your GenAI or LLM application, based on OpenTelemetry
https://github.com/traceloop/openllmetry

76 A complete guide to LLM observability with OpenTelemetry and ...
https://grafana.com/blog/2024/07/18/a-complete-guide-to-llm-observability-with-opentelemetry-and-grafana-cloud/

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

83 84 GitHub - Farama-Foundation/chatarena: ChatArena (or Chat Arena) is a Multi-Agent Language Game Environments for LLMs. The goal is to develop communication and collaboration capabilities of AIs.
https://github.com/Farama-Foundation/chatarena