Полное описание бота: версия 14

Это подробное описание всей системы бота для Telegram. Я постарался сделать текст максимально понятным для человека, который не является экспертом в программировании, но хочет понять, как всё работает. Я объясню каждый шаг простыми словами, как будто рассказываю другу, и укажу конкретные имена функций и методов из кода (например, `fetch_free_llms()` или `db.add_to_queue()`), чтобы было ясно, где это реализовано. Если встречу технический термин, то сразу объясню, что он значит.

Система — это умный чат-бот, который работает в Telegram. Он отвечает на сообщения пользователей, используя искусственный интеллект (большие языковые модели, или LLM, вроде тех, что в ChatGPT). Бот может "помнить" прошлые разговоры через специальную базу данных (RAG), обрабатывать много сообщений одновременно (асинхронно, то есть не ждёт, пока одно закончится, чтобы начать другое), и даже сам начинать разговор, если в чате тихо. Всё управляется через граф (как схема шагов) в библиотеке LangGraph, а настройки хранятся в файле .env (это файл с параметрами, вроде ключей API).

Бот экономит ресурсы (токены — это "единицы" для запросов к LLM), выбирая лучшую модель для каждого пользователя и иногда проверяя все модели заново. Есть интерфейс (UI) на Tkinter — простое окошко для контроля, где можно ставить паузу или добавлять пользователей в чёрный список.

Ключевые компоненты системы: что внутри и как они связаны

Давайте разберём систему по частям, как конструктор Lego. Каждая часть — это модуль (файл с кодом), и я укажу, где он находится.

1. **Telegram-бот (в файле main_flow.py)**:
   - Это "входная дверь" системы. Бот слушает сообщения в Telegram (использует библиотеку python-telegram-bot для асинхронного чтения — метод run_polling). Когда приходит сообщение, функция `telegram_handler(update, context)` проверяет, не в чёрном списке ли пользователь (вызывает `await db.is_blacklisted(user)`). Если всё ок, добавляет сообщение в очередь (асинхронно, через `await db.add_to_queue(user, message, chat_id)` — chat_id нужно, чтобы потом ответить в правильный чат). Если пауза включена (переменная paused), то только сохраняет в RAG (вызов `await rag.upsert(...)`), без обработки.

2. **Очередь сообщений (в файле sqlite.py, таблица queue)**:
   - Это как список задач для бота — сообщения ждут своей очереди. Очередь асинхронная (использует библиотеку aiosqlite, чтобы не блокировать другие процессы). Функции: `await db.add_to_queue(user, message, chat_id)` — добавляет, `await db.get_queue_item()` — берёт первое, `await db.remove_from_queue(item_id)` — удаляет после успешной отправки ответа в Telegram (чтобы не потерять, если отправка провалится).

3. **Граф обработки сообщений (LangGraph в файле main_flow.py, функция create_workflow)**:
   - Это "мозг" бота — схема шагов (ноды), как в flowchart. Каждое сообщение проходит через шаги: очистка и перевод, выбор роли, проверка роли, поиск контекста в RAG, выбор модели LLM, проверка на неактивность, генерация ответа, проверка качества ответа, сохранение в RAG, посылка ответа в телеграмм.. Если что-то не так, есть условные переходы (conditional edges). Граф компилируется с сохранением состояний в checkpoints.db (библиотека SqliteSaver).

4. **Менеджер LLM (файл llm.py, класс LLMManager)**:
   - Общается с сервисом OpenRouter (асинхронно, через aiohttp). Функции: `await invoke_llm(model, prompt, ...)` — базовый запрос к LLM с логированием в БД (`await db.log_llm_call(...)`). `await determine_role(message, model)` — выбирает роль. `await get_valid_role_score(role, message)` — проверяет роль множественными судьями (итеративно, пока score >= ROLE_SCORE_THRESHOLD, обновляет average_score). `await generate_responses_parallel(...)` и `await score_responses_parallel(...)` — параллельно генерируют и оценивают ответы (asyncio.gather).
Проверка качества ответа

5. **RAG-менеджер (файл rag.py, класс RAGManager)**:
   - Это "память" бота — векторная база данных (Pinecone или Qdrant). Сохраняет сообщения и ответы как векторы (эмбеддинги через SentenceTransformer).
Есть особенность работы с Pinecone - ращмер эмбеддинга модет быть 384 *по умолчанию) или 768. У Pinecone есть встроенный эмбеддинг, там есть понятие namespace.
У Qdrant идет локальная реализация, нет понятия namespace но есть индексируемые разделы.
Для единообразия я думаю о использовании раздела nick_name как эквивалента namespace.
 Функция `await upsert(text, namespace, metadata)` — добавляет запись с метаданными (original_text, translated_text, clean_text, timestamp, author, is_bot, namespace). Для сообщений из Telegram: author = user (nick_name), namespace = user. `await query(query_text, namespace)` — ищет похожие записи (для user и глобального "0").
translate_text лучше добавлять в новую запись, потому что Pinecone поддерживает только индекс, эмбеддинг и текст, и есть раздел metadata в который только можно помещать nick_name, author, is_bot, time and date). тут nick_name = namespace, author ={nick_name , name_of_bot}

6. **База данных SQLite (файл sqlite.py, класс Database)**:
   - Хранит всё: модели (`models`), скоринги (`scores` с average_score и count), историю (`history`), логи (`llm_message_log`), очередь (`queue` с chat_id), настройки (`settings`). Все методы асинхронные (await ...). Например, `await set_model_score(user, model, score_dat)` — обновляет average по формуле (old_avg * N + score_dat) / (N+1). `await get_setting(key)` и `await set_setting(key, value)` — для флагов, как fixed_llm_{user} или global_message_count.

7. **Интерфейс пользователя (UI в файле main_flow.py, класс UI)**:

- Окошко на Tkinter (запускается в отдельном потоке). Показывает логи (`add_log`), входящие сообщения (`add_incoming`), ответы (`add_response`). Кнопки: чекбокс для очистки (set_clean — меняет global clean_messages), добавление в чёрный список (add_blacklist — `db.set_setting(...)`), установка оператора (set_operator), пауза (toggle_pause), сброс сессий (reset_sessions — удаляет fixed_llm_* из settings).

8. **Утилиты (файл utils.py)**:
- Общие функции: `load_env()` — загружает .env, `fetch_free_llms()` — список моделей, `translator.translate_to_english(text)` — перевод, `clean_message(message)` — очистка, `check_inactivity(last_activity)` — проверка неактивности, `logger.info/error` — логи.

9. **Дополнительный модуль для документов (файл doc_processing.py)**:
- Отдельная утилита (запускается вручную). Pull'ит репозиторий Git (clone_or_pull_repo), проверяет commit (get_latest_commit vs db.get_setting('doc_commit')), чанкует документы (process_documents с RecursiveCharacterTextSplitter), upsert в RAG (namespace="0", author=REPO_OWNER). До этого модно конверитровать все документы из разных форматов в один, напримр с помощьбю библиотеки Doclig

Как работает бот: шаг за шагом (общая логика запуска и работы)

Представьте, что бот — это конвейер в фабрике: сообщение приходит, проходит этапы, выходит ответ. Сообщения обрабатываются одно за другим, синхронно. Но сама обработка сообщения происходит асинхронно Чтобы обработка одного сообщения не слишком затягивалась.

1. Запуск программы (в файле main_flow.py, блок if __name__ == "__main__")
- Создаётся объект базы данных: `db = Database()` (для UI, синхронно, но методы асинхронные).
- Создаётся UI: `ui = UI(db)` — окошко с текстами и кнопками.
- UI запускается в отдельном потоке: `threading.Thread(target=ui.run).start()` — чтобы не блокировать основной цикл.
- Запускается асинхронный цикл: `asyncio.run(main())`.

2. Инициализация в main() (асинхронно)
- Создаётся db: `db = Database()`, await `db.init_tables()` — создаёт таблицы в bot.db (модели, скоринги с average_score, история, логи, очередь с chat_id, настройки).
- Создаётся rag: `rag = RAGManager()` — загружает эмбеддер, подключается к Pinecone/Qdrant.
- Создаётся llm: `llm = LLMManager(db)`.
- Загружаются модели: `models = fetch_free_llms()` — запрос к OpenRouter API или fallback.
- Сохраняются модели: await `db.set_free_llms(models)` — сбрасывает все is_active=0, добавляет новые (если нет в БД), активирует существующие из списка (UPDATE is_active=1).
- Создаётся граф: `app = create_workflow(db, rag, llm)` — добавляет ноды (translate_message, judge_role и т.д.), ребра, conditional (needs_rejudge на "RESCORE").

- Запускается Telegram-бот: `application = ApplicationBuilder().token(TELEGRAM_BOT_TOKEN).build()`, добавляет хэндлер `telegram_handler`, await initialize/start/polling.

3. Чтение сообщений из Telegram (функция telegram_handler, асинхронно)
- Получает update (сообщение).
- Извлекает user (username), message (text), chat_id.
- Если user в чёрном списке (await `db.is_blacklisted(user)`) — игнорирует.
- Добавляет в UI: `ui.add_incoming(user, message)`.
- Если пауза (paused): await `rag.upsert(message, user, {metadata с original_text, author=user, is_bot=0})` — сохраняет, но не обрабатывает.
- Иначе: Если очистка (clean_messages) — `clean_message(message)` (удаляет emoji и символы).
- Добавляет в очередь: await `db.add_to_queue(user, message, chat_id)`.

4. Основной цикл обработки в main() (while running, асинхронно)
- Если пауза — await asyncio.sleep(1), пропускает.
- Берёт из очереди: `item = await db.get_queue_item()` (id, user, message, timestamp, chat_id).
- Если есть: Создаёт state с user, message, chat_id.
- Обрабатывает: await `app.ainvoke(state)` — проходит граф (ноды ниже).
- После: response = result['response'].
- Отправляет: await `application.bot.send_message(chat_id=chat_id, text=response)`.
- Если успешно: `ui.add_response(user, response)`, await `db.remove_from_queue(item_id)`.
- Если ошибка: log_error, не удаляет (для retry).
- Если очереди нет: await asyncio.sleep(1) — здесь может сработать proactive_check в графе (проверяет absolute время, не зависит от sleep).

5. Обработка в графе LangGraph (app.ainvoke, асинхронно)
- **translate_message**: Очищает (если clean_messages), переводит (`translator.translate_to_english(clean_msg)`), сохраняет в history (`await db.add_history(...)`), upsert в rag с metadata (original_text, clean_text, translated_text, author=user, is_bot=0, namespace=user). Обновляет last_activity.
- **judge_role**: Выбирает роль: await `llm.determine_role(translated_message, default_model)`.
- **validate_role**: Проверяет: await `llm.get_valid_role_score(role, translated_message)` — множественные судьи (итеративно по active_llms, пока score >= ROLE_SCORE_THRESHOLD, обновляет average_score для судьи). Если < порога — role="RESCORE".
- Conditional needs_rejudge: Если "RESCORE" — назад к judge_role, иначе retrieve_rag.
- **retrieve_rag**: await `rag.query(...)` — ищет контекст, upsert translated_message.
- **select_llm**: fixed_llm = await `db.get_setting(f"fixed_llm_{user}")`. Иначе: global_count = await `db.get_setting("global_message_count", 0)`. Если count % FULL_SCORING_INTERVAL == 0 — models = все active_llms (полный скоринг). Иначе топ-N + outsider. await `db.set_setting("global_message_count", count + 1)`.
- **proactive_check**: Если неактивность (check_inactivity) — роль "AGITATOR".

- Conditional is_proactive: Всегда к generate_response.
- **generate_response**: Генерирует parallel (await `llm.generate_responses_parallel(...)`), скорит (await `llm.score_responses_parallel(...)`), выбирает best, фиксирует (`await db.set_setting(f"fixed_llm_{user}", best_model)`), сохраняет в history и rag, обновляет average_score (`await db.set_model_score(user, best_model, score)`).
- **log_and_save**: logger.info, await `db.set_model_score(...)` (ещё раз, но уже после всего).

6. Проактивные действия (в proactive_check)
- Глобальная last_activity обновляется при сообщениях. Если time.time() > last_activity + random (INACTIVITY_MIN-MAX мин *60) — меняет роль на "AGITATOR", сообщение на провокацию.

7. Сессии и экономия (в select_llm и generate_response)
- Фиксирует лучшую модель на сессию (fixed_llm_{user}). Сбрасывается в UI (reset_sessions — удаляет LIKE 'fixed_llm_%').
- Полный скоринг раз в FULL_SCORING_INTERVAL сообщений (глобально, не per user).

8. Логирование и сохранение
- Ошибки: log_error(e, context).
- LLM-вызовы: await `db.log_llm_call(...)`.
- История: await `db.add_history(...)`.
- RAG: await `rag.upsert(...)` с полными metadata.
- Скоринги: await `db.set_model_score(...)` — вычисляет average после каждого скоринга.

9. UI-методы (в классе UI)
- set_clean: Меняет clean_messages.
- add_blacklist: `db.set_setting("blacklist_{user}", True)`.
- set_operator: `db.set_setting("operator", value)`.
- toggle_pause: Переключает paused.
- reset_sessions: Удаляет fixed_llm_*.
- add_log/incoming/response: Добавляет текст в окошки.

Система асинхронна (asyncio для параллелизма), экономит токены (топ-N моделей + полный скоринг редко), поддерживает до 50 LLM. Для запуска: pip install -r requirements.txt, настрой .env, запусти python main_flow.py. Если вопросы — смотри логи в bot_system.log.