# 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.