
Инженер
Создание AI-агента на LangChain и OpenRouter

Инженер
Создание AI-агента на LangChain и OpenRouter

http://www.litres.ru/pages/biblio_book/?art=73084753
SelfPub; 2026

Аннотация

Книга пошагово учит создавать интеллектуальных AI-агентов с использованием LangChain (для проектирования цепочек рассуждений и работы с LLM) и OpenRouter (для доступа к множеству языковых моделей через единый API).

Вы освоите:

настройку окружения и интеграцию LangChain с OpenRouter;

проектирование агентов с памятью и инструментами (поиск, API, базы данных);

работу с промптами, цепочками и кастомными действиями;

развёртывание и оптимизацию агентов для реальных задач.

Практические примеры включают:

чат-бота с доступом к актуальной информации;

агента для анализа документов и генерации отчётов;

систему автоматизации рутинных операций.

Для кого книга:

разработчики, желающие внедрить AI в свои проекты;

аналитики и инженеры данных, ищущие способы автоматизации;

стартаперы, создающие AI-продукты.

Результат: вы сможете проектировать и запускать AI-агентов, решающих прикладные задачи.

Содержание

Инженер Создание AI-агента на LangChain и OpenRouter

Глава 1. Введение и настройка окружения

Введение и настройка окружения

Добро пожаловать в практическое руководство по созданию интеллектуальных агентов на стыке мощи крупных языковых моделей и гибкости современных фреймворков. Эта книга посвящена тому, как объединить доступ к сотням моделей через единый концентратор OpenRouter и модульную экосистему LangChain для построения надежных, autonomных и полезных агентов. Здесь мы не просто разберем теорию, а последовательно пройдем все этапы: от настройки окружения и понимания архитектуры до реализации сложных сценариев с планированием, инструментами, долгосрочной памятью, устойчивостью к ошибкам и безопасностью. Если ваша цель – превратить потенциал современного ИИ в рабочий инструмент, который выполняет задачи, взаимо-

действует с внешним миром и масштабируется в продакшене, вы на верном пути.

Зачем нужен такой подход. Мир LLM переживает взрывной рост: появляются новые модели, улучшаются возможности, снижается стоимость. Но простой вызов модели с промптом уже недостаточен для решения реальных задач. Реальность требует структуры: агент должен разбивать сложную задачу на подзадачи, выбирать инструменты, запоминать контекст, корректировать поведение и взаимодействовать с API, базами данных и внешними сервисами. OpenRouter дает вам ключи к множеству моделей через единый интерфейс, устраняя фрагментацию провайдеров и упрощая эксперименты. LangChain предоставляет строительные блоки для цепочек, агентов, инструментов и памяти, позволяя собирать из этих блоков надежные системы. Вместе они образуют связку, которая делает разработку практичной, гибкой и воспроизводимой.

Ключевые идеи и обещания книги. Мы будем подходить к созданию агентов системно: проектировать логику, выбираем модели под задачу, настраиваем стабильные цепочки, встроим инструменты, отладим ошибки и подготовим к работе под нагрузкой. Вы увидите, как превратить «разговорную» модель в исполнителя команд, который может проводить вычисления, искать данные, хранить историю и коррек-

тировать свои действия. Мы будем использовать практические примеры: от простых агентов-помощников до многопроцессных конвейеров с очередями и дашбордами. Особое внимание уделим качеству промптов, стабильности работы и безопасности: как не допустить утечек данных, как ограничить действия агента и как контролировать ресурсы.

Структура книги. Первая часть посвящена основам: знакомству с OpenRouter, LangChain и настройке окружения. Вторая часть посвящена архитектуре агента: циклу размышлений, планированию, выбору инструментов, обработке ошибок и рефлексии. Третья часть фокусируется на интеграциях: базы данных, файловые системы, веб-скрапинг, вызовы внешних API, обработка документов. Четвертая часть – про производительность и надежность: кеширование, параллелизация, логирование, метрики, тестирование. Пятая часть – про безопасность и соответствие требованиям: валидация ввода, изоляция исполнения, ограничение прав, мониторинг. Шестая часть – продвинутые сценарии: мультиагентные системы, обучение на собственных данных, настройка под конкретные вертикали. В каждой главе есть код, примеры конфигураций, чек-листы и практические советы.

Ожидания и целевая аудитория. Книга рассчитана на разработчиков, инженеров данных, продакт-менеджеров и технических лидеров, которые хотят создавать рабочие AI-аген-

ты. Предполагается знакомство с Python и базовыми понятиями API, но сложные моменты мы объясняем по ходу дела. Если вы новичок, начинайте с главы по настройке и медленно продвигайтесь, повторяя примеры. Если вы опытный инженер, вам будут интересны главы про масштабирование, надежность и безопасность – там мы затрагиваем архитектурные решения для продакшена.

Основные понятия. Агент – программа, которая получает цель, разбивает её на шаги, использует инструменты и принимает решения на основе выводов модели. Инструмент – внешняя функция или API, которую агент может вызвать для получения информации или выполнения действия. Цепочка – последовательность шагов, которые выполняются для получения результата. Память – способ сохранять контекст между вызовами, чтобы агент помнил предыдущие действия и пользовательские данные. Планировщик – компонент, который формирует план действий и корректирует его по мере поступления новой информации. Рефлексия – механизм, с помощью которого агент анализирует свои ошибки и улучшает дальнейшие шаги. OpenRouter – агрегатор моделей, предоставляющий доступ к разным LLM через единый API. LangChain – фреймворк для построения цепочек, агентов и инструментов.

Почему OpenRouter и LangChain. OpenRouter позволяет

оперативно переключаться между моделями, сравнивать их производительность, управлять квотами и токенами без кастомной интеграции под каждый провайдер. Это экономит время и дает гибкость: можно использовать легкие модели для простых задач и мощные модели для сложных размышлений, распределять нагрузку и бюджет. LangChain, в свою очередь, предоставляет богатую экосистему: загрузчики документов, сплиттеры, векторные хранилища, инструменты, готовые агенты и интеграции. Он позволяет описывать логику в декларативном стиле, но при этом оставляет контроль над низкоуровневыми деталями. Связка OpenRouter + LangChain дает вам универсальность моделей и структуру для их эффективного использования.

Первые шаги, которые мы сделаем вместе. Мы настроим виртуальное окружение, установим зависимости, заведем ключи OpenRouter и проверим подключение к модели. Затем создадим минимального агента: примем запрос от пользователя, выберем инструмент, получим результат и отформатируем ответ. После этого добавим память, чтобы агент мог работать в диалоге, и расширим функционал: добавим планировщик и рефлексию. Мы подключим внешние инструменты – например, запросы к веб–API, поиск по базе данных, чтение файлов. Научимся обрабатывать ошибки, ограничивать число шагов, логировать промежуточные состояния и тестировать цепочки. И наконец, подготовим про-

дакшен-версию: контейнеризацию, переменные окружения, обработку лимитов, мониторинг и дашборд.

Архитектурные рекомендации. Агент должен быть профилирован: четко определите его роль, допустимые инструменты и границы поведения. Используйте модульность: отдельные модули для промптов, инструментов, памяти, планировщика и мониторинга. Сделайте промпты управляемыми через шаблоны и версионируйте их. Предусмотрите механизмы fallback: если основная модель не отвечает, переключитесь на более легкую или используйте заглушку. Внедрите рефлексию: на каждом шаге анализируйте, удалось ли достичь цели, и корректируйте план. Собирайте телеметрию: токены, время ответа, количество вызовов инструментов, ошибки. Соблюдайте безопасность: валидируйте ввод, ограничивайте доступ к инструментам, изолируйте выполнение, храните секреты в защищенном хранилище.

Типичные сценарии использования. Автоматизация рутинных задач: ответы на запросы пользователей, классификация тикетов, составление отчетов. Аналитика: извлечение данных из документов, обобщение, построение выдержек и ответов с ссылками. Работа с данными: запросы к CRM, обновление записей, создание задач в таск-менеджерах. Разработка: генерация кода по описанию, отладка, документирование. Исследование: поиск информации, реферирование,

проверка гипотез. Обучение: создание обучающих материалов, тестирование знаний, персонализация.

Ограничения и риски. Модели могут галлюцинировать, инструменты могут возвращать неожиданные результаты, внешние API могут падать. Важно проектировать систему с учетом этих рисков: добавлять валидацию результатов инструментов, просить модель проверять свои выводы, ограничивать число попыток, поддерживать человеко-вмешательство в критических сценариях. Не доверяйте агенту выполнять необратимые действия без подтверждения. Используйте песочницу для опасных операций. Соблюдайте регуляторные требования при работе с персональными данными.

Чек-лист успеха на старте. Определите цель агента и критерии успеха. Выберите стартовую модель через OpenRouter. Подготовите ключи API и переменные окружения. Напишите простой цикл: запрос -> планировщик -> выбор инструмента -> выполнение -> ответ. Добавьте память для диалога. Внедрите логирование и базовую рефлексию. Протестируйте на 10–20 разнообразных запросах. Замерьте метрики: точность, токены, время, стоимость. Сделайте бэкап промптов и конфигураций. Начните с ограниченного круга пользователей и соберите обратную связь.

Тонкости формулировок. Хороший промпт – это не про-

сто вопрос, а инструкция, которая задает роль, формат вывода, критерии принятия решений и ограничения. Мы будем много работать с шаблонами промптов, чтобы агент четко следовал схеме: структурировать мысль, определить план, выполнить шаги, проверить результат. В LangChain есть удобные примитивы для этого: системные сообщения, few-shot примеры, разделение на шаги. Мы будем использовать их для повышения стабильности.

План работы над ошибками. С самого начала мы заложим практику: логирование всех промежуточных состояний, сохранение «хлебных крошек» по пути агента, трассировку вызовов инструментов, трекинг токенов. Если агент зацикливается, мы введем ограничение на число итераций. Если ответ не соответствует формату – валидация и повторный запрос с уточнением. Если инструмент падает – fallback и альтернативный способ решения. Мы будем строить устойчивую систему, а не просто «рабочий скрипт».

Экономика. Стоимость токенов и вызовов API влияет на архитектуру. Мы покажем, как выбирать модель под задачу: тяжелые модели для планирования и рефлексии, легкие для форматирования и простых ответов. Обсудим кеширование повторяющихся запросов, батчинг, снижение числа вызовов за счет более четких промптов. OpenRouter позволяет легко мониторить расходы – мы интегрируем эти метрики в нашу

систему.

Качество и UX. Ответы агента должны быть понятными, структурированными и полезными. Мы будем учить форматировать выводы, прикреплять источники, четко обозначать неопределенности. Пользователь должен видеть, почему агент выбрал тот или иной инструмент, и иметь возможность запросить подробности. Мы добавим режимы «показать мысль» и «краткий ответ», а также подтверждение для опасных действий.

Производительность. Мы рассмотрим асинхронность, очереди задач, балансировку нагрузки между моделями, ограничение параллельных вызовов, таймауты. Для долгих операций мы сделаем фоновые задачи и статусы выполнения. Для интерактивных сценариев – streaming ответы и прогресс-бары. Мы подойдем к этому как инженеры: с профилированием, бенчмарками и оптимизациями.

Безопасность и соответствие. Мы будем избегать хранения чувствительных данных в промптах, использовать переменные окружения и секрет-менеджеры, валидировать ввод и вывод, ограничивать доступ к инструментам. Мы обсудим политики, когда необходимо участие человека, и как аудировать действия агента. Это важно не только для соответствия регламентам, но и для выстраивания доверия пользователей.

Культура разработки. Внедрим ревью промптов, версионирование конфигураций, тесты для цепочек, интеграционные тесты для инструментов, дашборды для мониторинга. Мы будем подходить к агенту как к системе, требующей поддержки и улучшений. Инструменты, которые мы используем, будут открытыми и стандартными, чтобы вы могли легко адаптировать их под свой стек.

Давайте резюмируем. Ваш агент будет получать задачу, разбивать её на шаги, выбирать инструменты через LangChain, обращаться к моделям через OpenRouter, хранить историю, корректировать себя по мере работы и возвращать структурированный результат. Мы будем строить его так, чтобы он был стабильным, безопасным, понятным и экономичным. Каждый этап будет подкреплен примерами кода, конфигурациями и чек-листами. Итогом станет не просто прототип, а готовая к использованию система, которую можно масштабировать и поддерживать.

Начальная дорожная карта. После этой главы мы сразу перейдем к настройке окружения и первого запуска. В ближайших главах построим минимального агента, добавим память и инструменты, затем улучшим стабильность и надежность. Параллельно мы будем отвечать на вопросы: как выбирать модель, как писать эффективные промпты, как тестировать

и как готовить систему к продакшену. Вы сможете повторять примеры шаг за шагом и в конце получить работающего агента под ваши задачи.

Поехали. Следующий шаг – подготовка окружения, установка зависимостей и первый запрос к модели через OpenRouter в контексте LangChain.

Переходим к практической части. Начнем с подготовки рабочего места: установим Python, создадим виртуальное окружение, установим необходимые библиотеки, настроим переменные окружения и проверим базовый вызов модели через OpenRouter в LangChain. Мы будем использовать современные подходы, чтобы сразу заложить основу для надежного агента.

Установка Python и виртуальное окружение. Рекомендуем версию Python 3.10 или новее. Проверьте версию:

python –version

Если у вас несколько версий, используйте явный вызов через python3. Создайте каталог проекта и активируйте виртуальное окружение:

mkdir ai_agent_project

cd ai_agent_project

python -m venv .venv

Активация в Linux/macOS:

source .venv/bin/activate
Активация в Windows:
.venv\Scripts\activate
Обновите pip:
pip install –upgrade pip

Установка LangChain и интеграций. Начнем с основных пакетов:

pip install langchain langchain-core langchain-community

Пакет langchain-community содержит множество инструментов и интеграций. Для работы с OpenRouter мы будем использовать стандартные HTTP-вызовы или готовый клиент, если он доступен в экосистеме. Для удобства также установим поддержку асинхронности и логирования:

pip install httpx tenacity pydantic dotenv

Для локального хранения секретов и конфигурации:

pip install python-dotenv

Если вы планируете использовать векторные хранилища и обработку документов, добавьте:

pip install langchain-text-splitters langchain-chroma sentence-transformers

Для интерактивных experiments и визуализации логов полезны:

pip install rich streamlit

Для тестов и статического анализа:

pip install pytest black isort

Настройка ключей OpenRouter. Зарегистрируйтесь на openrouter.ai, создайте ключ API в настройках. Откройте файл .env в корне проекта и добавьте:

```
OPENROUTER_API_KEY=your_key_here
OPENROUTER_BASE_URL=https://openrouter.ai/api/v1
```

Это базовые переменные. Можно также добавить:

```
OPENROUTER_DEFAULT_MODEL=anthropic/
claude-3.5-sonnet
OPENROUTER_TIMEOUT=60
```

При желании добавьте SITE_URL и SITE_NAME – они иногда используются провайдерами для идентификации запросов. Никогда не коммитьте .env в репозиторий. Добавьте .env в .gitignore:

```
.env
.env.local
secrets.*
```

Тестовый скрипт для проверки подключения. Создадим минимальный скрипт, который отправляет запрос через OpenRouter и получает ответ. Используем прямой HTTP-вызов, чтобы убедиться, что ключ работает. Файл check_openrouter.py:

```
import os
import httpx
from dotenv import load_dotenv
```

```
load_dotenv()

API_KEY = os.getenv("OPENROUTER_API_KEY")
BASE_URL = os.getenv("OPENROUTER_BASE_URL",
"https://openrouter.ai/api/v1")
MODEL =
os.getenv("OPENROUTER_DEFAULT_MODEL",
"anthropic/claude-3.5-sonnet")

def test_call():
headers = {
"Authorization": f"Bearer {API_KEY}",
"Content-Type": "application/json",
"HTTP-Referer": "https://example.com",
"X-Title": "AI Agent Test"
}
payload = {
"model": MODEL,
"messages": [
{"role": "system", "content": "You are a helpful assistant."},
{"role": "user", "content": "Привет! Напиши слово 'тест'
и цифру 42."}
],
"max_tokens": 100
}
```

try:
resp = httpx.post(f"{BASE_URL}/chat/completions", json=payload, headers=headers, timeout=60)
resp.raise_for_status()
data = resp.json()
content = data["choices"][0]["message"]["content"]
print("Ответ модели:", content)
print("Проверка пройдена!")
except Exception as e:
print("Ошибка:", e)

if __name__ == "__main__":
test_call()
Запустите:
python check_openrouter.py
Если вы видите ответ – ключ настроен верно.

Интеграция с LangChain. LangChain позволяет описывать цепочки и агенты декларативно. Для OpenRouter можно использовать класс ChatOpenAI с переопределением базового URL, либо любой другой провайдер, совместимый с OpenAI-форматом. Пример инициализации модели через LangChain:

from langchain_core.messages import HumanMessage, SystemMessage
from langchain_community.chat_models import

ChatOpenAI

```
import os
from dotenv import load_dotenv

load_dotenv()

model = ChatOpenAI(
model="anthropic/claude-3.5-sonnet",
openai_api_key=os.getenv("OPENROUTER_API_KEY"),
openai_api_base=os.getenv("OPENROUTER_BASE_URL"
temperature=0.3,
max_tokens=500
)

messages = [
SystemMessage(content="Ты полезный ассистент. Отвечай кратко и по делу."),
HumanMessage(content="Скажи слово 'успех' и текущий год.")
]

result = model.invoke(messages)
print(result.content)
```

Если по какой-то причине прямой класс ChatOpenAI не работает с OpenRouter, можно создать кастомный LLM-обертку через RunnableLambda или использовать прямой

HTTP-вызов внутри LangChain через LLM-интеграцию. Главное – сохранить формат чата и авторизацию.

Первый агент на LangChain. Построим минимального агента, который просто вызывает модель и форматирует ответ. Этот агент будет базовым каркасом для будущих расширений. Файл simple_agent.py:

```
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_community.chat_models import ChatOpenAI
from dotenv import load_dotenv
import os

load_dotenv()

model = ChatOpenAI(
model="anthropic/claude-3.5-sonnet",
openai_api_key=os.getenv("OPENROUTER_API_KEY"),
openai_api_base=os.getenv("OPENROUTER_BASE_URL"
temperature=0.1,
max_tokens=500
)

def run_agent(query: str):
system = SystemMessage(content="Ты – агент, который от-
```

вечает точно и по факту. Форматируй ответ структурированно.")
    user = HumanMessage(content=query)
    response = model.invoke([system, user])
    return response.content

    if __name__ == "__main__":
    print(run_agent("Перечисли три цвета радуги через запятую."))

Запустите и убедитесь, что ответ приходит. Этот простой цикл – ядро будущего сложного агента.

Установка дополнительных инструментов для LangChain. Чтобы агент мог выполнять действия, нужны инструменты. Примеры:

pip install duckduckgo-search # поиск
pip install wikipedia # энциклопедия
pip install sqlalchemy # базы данных
pip install beautifulsoup4 # парсинг веб-страниц

Это расширит возможности агента. Мы подключим их позже, когда будем строить цикл «мыслить – действовать – проверять».

Проверка лимитов и политик. Убедитесь, что ваш аккаунт OpenRouter имеет доступ к выбранной модели. Если вы получаете ошибку 402 или 429, проверьте баланс и квоты. На-

стройте таймауты и повторные попытки с экспоненциальной задержкой. Для этого используем библиотеку tenacity:

```
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1, min=2, max=10))
def call_model_safe(payload, headers):
import httpx
resp = httpx.post("https://openrouter.ai/api/v1/chat/completions", json=payload, headers=headers, timeout=60)
resp.raise_for_status()
return resp.json()
```

Это повысит устойчивость к временным сбоям сети или API.

Логирование и отладка. Добавим простой логгер, чтобы видеть шаги агента. В Python стандартный logging подойдет:

```
import logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s – %(levelname)s – %(message)s")
logger = logging.getLogger("agent")
logger.info("Запуск агента")
```

Вставляйте логи в ключевых точках: вызов модели, вызов инструментов, ошибка, успешное завершение. Для красивого вывода можно использовать Rich:

from rich.console import Console
console = Console()
console.print("[bold green]Агент[/bold green] запущен")
Это упрощает отладку при работе в терминале.

Структура проекта. Рекомендуемая структура для масштабируемой разработки:
src/
__init__.py
llm.py # инициализация модели OpenRouter
tools.py # набор инструментов агента
memory.py # управление контекстом и памятью
planner.py # компонент планирования
reflexion.py # анализ ошибок и корректировка
agent.py # основной цикл агента
tests/
test_tools.py
test_agent.py
config/
prompts.yaml # шаблоны промптов
settings.py # настройки и переменные
.env
.gitignore
README.md
requirements.txt
Такой проект легко поддерживать, тестировать и депло-

ить.

Зависимости в requirements.txt. После установки всех пакетов создайте файл:

pip freeze > requirements.txt

Убедитесь, что в нем есть:

langchain
langchain-core
langchain-community
httpx
tenacity
python-dotenv
rich
duckduckgo-search
wikipedia
sqlalchemy
beautifulsoup4
sentence-transformers
langchain-chroma
streamlit (если нужен веб-интерфейс)

Это обеспечит воспроизводимость окружения.

Конфигурация через переменные окружения. Помимо ключей, добавьте флаги для режимов:

AGENT_DEBUG=true # включить подробные логи
AGENT_MAX_STEPS=10 # лимит шагов агента

AGENT_TOOL_TIMEOUT=30 # таймаут для инструментов

MODEL_TEMPERATURE=0.1 # температура по умолчанию

Используйте их в коде:

debug = os.getenv("AGENT_DEBUG", "false").lower() == "true"

if debug:

logger.setLevel(logging.DEBUG)

Это позволит включать детальные логи в dev и отключать в prod.

Первый запуск цепочки с инструментом. Соберем простую цепочку: модель -> инструмент поиска -> форматирование. Пример:

```
from langchain_community.utilities import DuckDuckGoSearchAPIWrapper
from langchain.agents import Tool
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_community.chat_models import ChatOpenAI
import os
from dotenv import load_dotenv

load_dotenv()
```

```
search = DuckDuckGoSearchAPIWrapper()
tool = Tool(name="search", func=search.run, description="Используй для поиска информации в интернете.")

model = ChatOpenAI(
model="anthropic/claude-3.5-sonnet",
openai_api_key=os.getenv("OPENROUTER_API_KEY"),
openai_api_base=os.getenv("OPENROUTER_BASE_URL"),
temperature=0.1
)

def simple_agent_with_tool(query: str):
# Сначала спросим модель, нужно ли искать информацию
sys_msg = SystemMessage(content="Ты агент. Если в запросе нужна актуальная информация, используй инструмент поиска. Форматируй ответ.")
user_msg = HumanMessage(content=query)
# В этом примере мы просто выполняем поиск и подаем результат в модель
search_result = tool.func(query)
augmented_messages = [
sys_msg,
HumanMessage(content=f"Вопрос: {query}\nРезультаты поиска:\n{search_result}\n\nСформулируй четкий ответ.")
```

```
]
response = model.invoke(augmented_messages)
return response.content

if __name__ == "__main__":
print(simple_agent_with_tool("Какая погода в Лондоне сегодня?"))
```

Этот подход имитирует «мысль – действие – ответ». Более продвинутые версии мы реализуем с помощью LangChain Agents в следующих главах.

Тестирование и валидация. Напишем простой тест для функции запуска агента:

```
# tests/test_agent.py
import sys
sys.path.append("src")
from simple_agent import run_agent

def test_run_agent():
res = run_agent("Скажи слово 'проверка'")
assert "проверка" in res.lower()
```

Вам понадобится pytest для запуска:

```
pytest tests/
```

Если тесты падают из-за проблем с сетью или лимитами, добавьте моки для вызовов API или используйте записанные ответы.

Безопасность и секреты. Никогда не храните ключи в коде. Используйте .env и системные переменные в продакшене. Если вы работаете в облаке, используйте менеджеры секретов: AWS Secrets Manager, Google Secret Manager, Azure Key Vault. При локальной разработке защищайте файл .env: права доступа 600 и исключение из бэкапов. В коде используйте валидацию, чтобы не передавать ключи в логах и сообщениях модели.

Проверка поддержки streaming. Для интерактивных сценариев полезен потоковый вывод. Если модель поддерживает streaming через OpenRouter, можно настроить постепенную выдачу токенов. В LangChain есть метод stream:

```
for chunk in model.stream([HumanMessage(content="Расскажи анекдот в три предложения.")]):
print(chunk.content, end="", flush=True)
```

Это улучшает восприятие пользователем длительных ответов.

Мониторинг и метрики. С самого начала собирайте метрики: число токенов, время выполнения, количество вызовов инструментов, количество итераций. Добавим простой трекер:

```
from dataclasses import dataclass
import time
```

```
@dataclass
class Metrics:
tokens_in: int = 0
tokens_out: int = 0
tool_calls: int = 0
duration_ms: int = 0

def track_metrics_start():
return {"start": time.time()}

def track_metrics_end(stats, metrics: Metrics):
metrics.duration_ms = int((time.time() – stats["start"]) * 1000)
return metrics
```

Заполнять поля tokens_in/tokens_out можно из ответов API, если модель возвращает usage. OpenRouter часто возвращает usage, если поддерживается провайдером. Мы будем использовать это для управления стоимостью.

Обработка ошибок и отказоустойчивость. Всегда оборачивайте вызовы модели и инструментов в try/except. При ошибке не прерывайте работу, а предлагайте альтернативу: повторить запрос, уточнить детали, переключить модель, использовать кешированный ответ. В LangChain есть встроенные механизмы повторных попыток и фолбэков – мы их до-

бавим позже. Сейчас заложим привычку логировать ошибки и возвращать пользователю дружелюбное сообщение.

Версионирование промптов. Создайте в config/prompts.yaml файл с шаблонами:

system_agent: "Ты агент. Форматируй ответ. Не придумывай факты."

system_reflexion: "Проверь, соответствует ли ответ цели. Если нет, предложи корректировку."

system_tool_guidance: "Определи, нужен ли инструмент. Если да, выведи только вызов в формате tool:имя_инструмента|параметры."

Загружайте их в коде:

```
import yaml
with open("config/prompts.yaml") as f:
prompts = yaml.safe_load(f)
system = prompts["system_agent"]
```

Так вы сможете менять поведение без переписывания кода.

Первые шаги пользователя. После настройки окружения рекомендуем выполнить три упражнения, чтобы закрепить понимание:

1) Отправьте запрос без инструментов и убедитесь, что ответ приходит. Измените температуру и сравните результаты. 2) Добавьте инструмент поиска и попросите агента най-

ти свежую информацию. Проверьте, что он использует результаты поиска. 3) Включите логирование и посмотрите, как формируются промежуточные сообщения. Попробуйте streaming-вывод для длинных ответов.

Когда переходить к следующей главе. Вы готовы к расширению агента, когда у вас:

– рабочее окружение и ключи;
– базовый вызов модели через LangChain;
– хотя бы один инструмент подключен;
– логирование и обработка ошибок;
– понимание формата промптов.

Заключение главы. Мы подготовили окружение для разработки AI-агента, настроили доступ к моделям через OpenRouter и создали базовые строительные блоки на LangChain. Вы получили минимального агента, способного отвечать на запросы, и набор инструментов для дальнейшего расширения. В следующих главах мы углубимся в архитектуру агента: как он мыслит, как выбирает инструменты, как хранит память и как корректирует свои действия. Мы заложим основы для надежного, масштабируемого и безопасного решения, которое можно адаптировать под любую задачу. А пока – проверьте, что все работает, и начните экспериментировать с простыми запросами. Успехов!

Глава 2. Установка и конфигурация LangChain и OpenRouter

Глава 2. Установка и конфигурация LangChain и OpenRouter

Добро плавать в практической разработке AI-агентов. Эта глава посвящена не только командам установки, но и пониманию экосистемы, выбору инструментов, настройке безопасного рабочего окружения и первой интеграции LangChain с OpenRouter. Мы пройдем весь путь от подготовки системы до написания первой цепочки, способной обращаться к моделям через единый API-интерфейс. Цель главы – создать надежный фундамент, на котором будут строиться сложные агенты.

Подготовка рабочего окружения

Прежде чем писать код, необходимо подготовить изолированную среду разработки. Работа с Python и его библиотеками требует осторожности: разные проекты могут зависеть от конфликтующих версий пакетов. Чтобы избежать проблем, мы будем использовать виртуальное окружение.

Если вы работаете на Linux или macOS:

1. Откройте терминал.
2. Создайте папку проекта: mkdir ai_agent_project && cd ai_agent_project.
3. Создайте виртуальное окружение: python3 -m venv venv.
4. Активируйте окружение: source venv/bin/activate.

Если вы работаете на Windows (PowerShell):
1. Создайте папку проекта: New-Item -ItemType Directory -Path "ai_agent_project"; Set-Location ai_agent_project.
2. Создайте виртуальное окружение: python -m venv venv.
3. Активируйте окружение: .\venv\Scripts\Activate.ps1. (Возможно, потребуется выполнить Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser для разрешения выполнения скриптов).

Если вы работаете на Windows (CMD):
1. md ai_agent_project && cd ai_agent_project.
2. python -m venv venv.
3. venv\Scripts\activate.

Внутри активированного окружения вы увидите префикс (venv) в начале строки терминала. Это означает, что все установленные пакеты будут изолированы от глобального пространства Python.

Установка Python и необходимых инструментов

Убедитесь, что у вас установлен Python версии 3.8 или выше. Рекомендуется версия 3.10 или 3.11 для лучшей совместимости с современными библиотеками AI. Проверить версию можно командой:

python –version

Обновление pip (пакетный менеджер) – обязательный шаг для избежания ошибок при установке сложных зависимостей:

pip install –upgrade pip

Для удобной работы с переменными окружения (API-ключами) установите python-dotenv:

pip install python-dotenv

Установка LangChain

LangChain – это экосистема, состоящая из множества пакетов. Начальная установка может сбить с толку из-за модульной структуры. Есть два пути: установка всех возможных компонентов сразу (не рекомендуется, так как создает тяжеловесную среду) или точечная установка необходимых модулей.

Для начала работы с OpenRouter и базовыми возможностями агента нам понадобится ядро библиотеки и клиент для работы с API.

Основной пакет:
pip install langchain

Важно понимать, что LangChain Core содержит базовые абстракции (цепочки, инструменты, агенты), но не содержит реализации подключения к конкретным LLM. Для работы с OpenRouter мы будем использовать стандартный клиент OpenAI, так как OpenRouter предоставляет совместимый API.

Также установим поддержку потоковой генерации (streaming) и логирование для отладки:
pip install langchain-core

В экосистеме LangChain 0.2.x и выше рекомендуется использовать пакет langchain-community для интеграций сторонних провайдеров, но для OpenRouter мы часто используем прямую совместимость с OpenAI SDK.

Для полноценной работы агента, который может использовать инструменты и планирование, установим дополнительные компоненты:
pip install langchain-openai # Если используем официальный клиент OpenAI как бэкенд для OpenRouter
pip install langgraph # Для построения агентов как графов

(современный подход)

pip install numpy # Часто требуется для обработки данных и векторных операций

pip install requests # Для прямых HTTP запросов если потребуется

Примечание: В книге мы будем использовать подход "агента на графах" (LangGraph), так как это наиболее гибкий и прозрачный способ управления состоянием агента, его памятью и логикой переходов между задачами.

Установка OpenRouter

OpenRouter – это агрегатор моделей. У него нет собственной библиотеки Python, который нужно устанавливать отдельно. Взаимодействие с ним происходит через стандартный API, совместимый с OpenAI, или через обычные HTTP-запросы.

Однако для удобства можно установить клиент OpenAI SDK, так как OpenRouter полностью эмулирует его интерфейс:

pip install openai

Это не является обязательным для LangChain (так как LangChain имеет встроенные адаптеры), но полезно для отладки и прямых запросов.

Получение API-ключа OpenRouter

Без ключа API вы не сможете делать запросы к моделям.

1. Перейдите на сайт openrouter.ai.
2. Зарегистрируйтесь или войдите в аккаунт.
3. Перейдите в раздел "Keys" (Ключи).
4. Нажмите "Create Key" (Создать ключ).
5. Дайте ключу имя (например, "My Agent Project") и скопируйте его.
6. ВАЖНО: Сразу же сохраните ключ в безопасном месте. OpenRouter не покажет его вам снова.

Настройка переменных окружения (Security Best Practices)

Никогда не храните API-ключи прямо в коде (внутри .py файлов). Если вы случайно загрузите код на GitHub, ваши ключи будут украдены ботами в течение минут. Правильный способ – использовать переменные окружения.

В корне вашего проекта создайте файл .env. Имя файла начинается с точки.

В Linux/macOS: touch .env

В Windows (PowerShell): New-Item -ItemType File -Path ".env"

Откройте этот файл в любом текстовом редакторе и до-

бавьте туда ваш ключ:

OPENROUTER_API_KEY=sk-or-v1-…
ваш_длинный_ключ…

Также добавьте ссылку на базовый URL API OpenRouter:

OPENROUTER_API_BASE=https://openrouter.ai/api/v1

Чтобы эти переменные загружались в Python-скрипт автоматически, используйте код:

```
from dotenv import load_dotenv
load_dotenv()
```

Теперь мы готовы к написанию кода.

Первая связка: LangChain и OpenRouter

OpenRouter работает как прокси, перенаправляя запросы к различным моделям (GPT-4, Claude, Mistral, Llama и т.д.). Ключевая задача – правильно настроить LangChain, чтобы он отправлял запросы именно на URL OpenRouter с вашим ключом.

Создадим файл main.py. Разберем два способа подключения: через ChatOpenAI (рекомендуемый) и через прямой вызов.

Способ 1: Использование ChatOpenAI (LangChain

OpenAI Wrapper)

Этот способ удобен, так как сохраняет все преимущества LangChain (простые промпты, форматирование сообщений, потоковую передачу данных).

Создайте файл setup_check.py и вставьте следующий код:

```
import os
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage

# Загружаем переменные из .env
load_dotenv()

# Получаем ключ и URL
api_key = os.getenv("OPENROUTER_API_KEY")
api_base = os.getenv("OPENROUTER_API_BASE")

if not api_key:
raise ValueError("API ключ не найден! Убедитесь, что вы создали .env файл и добавили OPENROUTER_API_KEY.")

# Инициализация модели через LangChain
# Обратите внимание на параметр base_url – это критически важно для OpenRouter
```

```
llm = ChatOpenAI(
model="openai/gpt-4o-mini", # Пример модели. Формат: провайдер/модель
openai_api_key=api_key,
openai_api_base=api_base,
temperature=0.7
)

# Простой запрос
message = HumanMessage(content="Привет! Ты работаешь через OpenRouter?")
response = llm.invoke([message])

print("Ответ агента:")
print(response.content)

# Проверка потоковой передачи (Streaming)
print("\n– Проверка потоковой передачи –")
for chunk in llm.stream([HumanMessage(content="Расскажи короткую историю о коте, который использует Python.")]):
print(chunk.content, end="", flush=True)
```

Объяснение кода:

1. load_dotenv() загружает ваши ключи.
2. ChatOpenAI – это класс-обертка над стандартным API

LLM.

3. Параметр openai_api_base указывает на адрес сервера OpenRouter.

4. openai_api_key использует ваш ключ авторизации.

5. Метод .invoke() делает единичный запрос и возвращает полный ответ.

6. Метод .stream() позволяет получать ответ по кусочкам (токенам), что критично для интерактивных интерфейсов.

Если при запуске python setup_check.py вы получили ответ от агента – поздравляю, связка работает!

Способ 2: Прямой вызов через LangChain (BaseChatModel)

Иногда требуется более низкоуровневый контроль или исключение лишних зависимостей. Мы можем использовать класс ChatOpenAI из langchain_openai, но настроить его вручную. Мы уже сделали это выше.

Важный нюанс: форматы моделей.

OpenRouter поддерживает тысячи моделей. Чтобы использовать конкретную, нужно знать ее идентификатор.

Примеры идентификаторов:

– openai/gpt-4-turbo-preview

– anthropic/claude-3-opus:beta

– meta-llama/llama-3-8b-instruct

– google/gemini-pro-1.5

Полный список доступен в документации OpenRouter. Если вы укажете несуществующую модель, OpenRouter вернет ошибку.

Конфигурация продвинутых параметров модели

При создании экземпляра ChatOpenAI можно передать множество параметров, влияющих на поведение агента:

```
llm = ChatOpenAI(
model="openai/gpt-4o-mini",
openai_api_key=api_key,
openai_api_base=api_base,
temperature=0.1, # Детерминированность (0 – строгий логик, 1 – творчество/хаос)
max_tokens=1024, # Ограничение на длину ответа
top_p=0.9, # Альтернативный метод фильтрации токенов (ядерная выборка)
frequency_penalty=0.5, # Штраф за повторение фраз (от -2.0 до 2.0)
presence_penalty=0.5, # Штраф за новые темы (от -2.0 до 2.0)
request_timeout=20 # Таймаут ожидания ответа в секундах
)
```

Настройка температуры для агентов

Для агентов, выполняющих задачи (Code Runner, Data Analyst), температура должна быть близка к 0 (например, 0.0 – 0.2). Это минимизирует галлюцинации и делает ответы предсказуемыми.

Для творческих агентов (Копирайтер, Поэт) используйте 0.7 – 1.0.

Создание цепочки (Chain) с использованием промптов

Теперь, когда мы умеем инициализировать модель, давайте создадим простейшую цепочку, которая берет входные данные, форматирует их и передает модели. Это основа любого агента.

Используем LangChain PromptTemplate для динамического создания промптов.

Импортируем необходимые модули:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
```

Создадим шаблон промпта:

```
prompt = ChatPromptTemplate.from_messages([
("system", "Ты полезный ассистент, который отвечает на русском языке."),
("human", "{input}") # {input} – это переменная, которую
```

мы будем подставлять
])

Соберем цепочку (Chain):
chain = prompt | llm | StrOutputParser()

Здесь используется оператор | (пайп), который объединяет компоненты:

1. prompt: создает структуру сообщения.
2. llm: отправляет в модель.
3. StrOutputParser(): извлекает текстовое содержимое из ответа модели (убирает внутреннюю структуру объекта Message).

Запуск цепочки:
result = chain.invoke({"input": "Что такое гравитация, объясни как пятилетнему ребенку?"})
print(result)

Использование StrOutputParser() важно для чистоты кода, так как возвращает строку вместо объекта AIMessage.

Интеграция инструментов (Tools) с OpenRouter

Агент – это не просто модель, отвечающая на вопросы. Агент способен вызывать функции, искать информацию и выполнять код. В LangChain инструменты (Tools) – это

функции, которые модель может вызвать по своему усмотрению.

Давайте создадим инструмент-калькулятор. Даже если модель GPT-4 умеет считать в уме, использование инструмента гарантирует точность (через выполнение Python-кода) и демонстрирует архитектуру агента.

Шаг 1: Определение функции и инструмента

```
from langchain_core.tools import tool

@tool
def multiply(a: int, b: int) -> int:
"""Умножает два целых числа."""
return a * b
```

Шаг 2: Подключение инструментов к модели

Агенту нужно знать, какие инструменты у него есть, и когда их использовать.

```
from langchain.agents import create_tool_calling_agent, AgentExecutor

# Список инструментов
tools = [multiply]
```

```
# Шаблон промпта для агента (он отличается от простого промпта)
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

agent_prompt = ChatPromptTemplate.from_messages([
("system", "Ты полезный математик. Используй доступные инструменты для точного решения задач."),
("human", "{input}"),
MessagesPlaceholder(variable_name="agent_scratchpad"), # Сюда будут записываться шаги мысли агента
])

# Создание агента
agent = create_tool_calling_agent(llm, tools, agent_prompt)

# Создание исполнителя
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# Запуск
response = agent_executor.invoke({"input": "Сколько будет 123 умножить на 456?"})
print(response['output'])
```

В этом коде:

1. create_tool_calling_agent создает агента, который умеет вызывать функции.

2. AgentExecutor управляет циклом "Решение -> Вызов инструмента -> Получение результата -> Формирование ответа".

3. Параметр verbose=True позволяет видеть в консоли логи вызова инструментов.

Сборка полноценного агента с OpenRouter и LangGraph

Классический AgentExecutor (используемый выше) постепенно устаревает. Современный стандарт – использование LangGraph. Это позволяет явно описывать граф состояний агента.

Давайте построим простого агента на LangGraph, который работает с OpenRouter.

Потребуется установка:
pip install langgraph

Создадим файл agent_graph.py:

```
import os
from dotenv import load_dotenv
from typing import Literal
from langchain_openai import ChatOpenAI
```

```
from langchain_core.tools import tool
from langgraph.prebuilt import create_react_agent
from langchain_core.messages import HumanMessage

# Настройка LLM (OpenRouter)
load_dotenv()
llm = ChatOpenAI(
model="openai/gpt-4o-mini",
openai_api_key=os.getenv("OPENROUTER_API_KEY"),
openai_api_base=os.getenv("OPENROUTER_API_BASE")
)

# Инструменты
@tool
def get_weather(city: str) -> str:
"""Получает текущую погоду в городе. Просто заглушка для примера."""
return f"В городе {city} солнечно и +25°C."

@tool
def get_stock_price(ticker: str) -> str:
"""Получает цену акции по тикеру."""
return f"Цена акции {ticker} сегодня выросла на 5%."

tools = [get_weather, get_stock_price]
```

```
    # Создание агента LangGraph (React Agent – Reasoning and
Acting)
    # Этот агент следует циклу: Thought -> Act -> Observation
-> Thought…
    graph = create_react_agent(llm, tools)

    # Интерфейс запуска
    def run_agent(question: str):
    inputs = {"messages": [HumanMessage(content=question)]}
    print(f"Вопрос: {question}")

    # Запуск графа потоком
    for event in graph.stream(inputs, stream_mode="values"):
    if "messages" in event:
    last_msg = event["messages"][-1]
    if isinstance(last_msg, HumanMessage):
    print(f"\nUser: {last_msg.content}")
    else:
    print(f"\nAgent: {last_msg.content}")

    if __name__ == "__main__":
    # Пример сложного запроса, требующего нескольких ин-
струментов
    run_agent("Какая погода в Лондоне и сколько стоит акция
AAPL?")
```

Разбор кода:

1. create_react_agent – это высокоуровневая функция, создающая готовый граф логики "Реакции" (ReAct). Она автоматически добавляет узлы для планирования и вызова инструментов.

2. graph.stream – позволяет в реальном времени выводить шаги агента.

3. В примере агент должен выбрать: сначала узнать погоду, потом цену акции, либо наоборот.

Почему LangGraph лучше?

– Прозрачность: вы видите каждый шаг.

– Контроль: можно вручную добавлять узлы (например, "Проверка безопасности", "Сохранение в БД").

– Поддержка потоковой передачи (streaming) агентов "из коробки".

Типичные ошибки при настройке и их решение

1. Ошибка 401 Unauthorized / Invalid API Key

Причина: Неверный ключ или отсутствие переменной окружения.

Решение: Проверьте файл .env. Убедитесь, что вы не скопировали лишние пробелы. Проверьте, что load_dotenv() вызывается до использования os.getenv().

2. Ошибка 404 Not Found (Model not found)

Причина: Неправильное имя модели в параметре model=.

Решение: Загляните в документацию OpenRouter. Имена чувствительны к регистру и слэшам. Попробуйте упростить имя (например, "openai/gpt-3.5-turbo").

3. Ошибка 402 Payment Required

Причина: На счету OpenRouter недостаточно средств.

Решение: Пополните баланс в личном кабинете OpenRouter. Некоторые модели (например, GPT-4) требуют предоплаты или имеют высокую стоимость запросов.

4. Ошибка AttributeError: 'NoneType' object has no attribute 'content'

Причина: Модель вернула пустой ответ или произошла ошибка парсинга.

Решение: Убедитесь, что вы используете StrOutputParser() или правильно извлекаете content из ответа.

5. Ограничения контекста (Context Window Exceeded)

Причина: История диалога стала слишком длинной для модели.

Решение: Используйте окна памяти (Memory) в LangChain, например ConversationBufferWindowMemory, чтобы сохранять только последние N сообщений.

Настройка продвинутых параметров OpenRouter

OpenRouter позволяет использовать специальные параметры через заголовки или передачу параметров. Например, можно принудительно использовать только определенные провайдеры или настроить "приставку" (prefix) для системных сообщений.

В LangChain при использовании ChatOpenAI мы можем передать дополнительные параметры через `extra_headers` или `extra_body`, если это поддерживается оберткой. Однако, базовые настройки (temperature, top_p) работают стандартно.

Если вам нужно использовать параметр `route`, который позволяет OpenRouter выбирать лучший провайдер автоматически, это часто происходит по умолчанию, если вы не указываете конкретную модель. Но для агентов рекомендуется жестко фиксировать модель (например, "openai/gpt-4o-mini") для стабильности работы инструментов.

Работа с системными промптами

Системный промпт – это инструкция, которая задает поведение всей цепочки. В OpenRouter это реализуется как роль `system`. В LangChain вы задаете это в ChatPromptTemplate.

Пример системного промпта для агента:

"Ты – helpful ассистент. Твоя задача – отвечать на вопросы пользователя точно и кратко. Если вопрос требует использования инструментов, ты обязан их использовать. Не придумывай факты, которые ты не можешь проверить."

Важно: Модели имеют разную чувствительность к системным промптам. Модели на базе Llama (Meta) часто требуют иного форматирования, чем GPT. OpenRouter старается стандартизировать это, но будьте готовы к экспериментам.

Модульная архитектура LangChain: что устанавливать?

В экосистеме LangChain происходит активное разграничение пакетов.

– `langchain-core`: Базовые абстракции. Обязателен.

– `langchain`: Старая "монолитная" библиотека. Сейчас часто не требуется, если не используете старые агенты.

– `langchain-community`: Интеграции с провайдерами (не OpenAI).

– `langchain-openai`: Специфичные интеграции для OpenAI (и OpenRouter).

– `langgraph`: Построение графов агентов.

Для этой книги мы рекомендуем структуру зависимостей:

```
langchain-core
```

```
langchain-openai
langgraph
python-dotenv
```

Если вы устанавливаете `langchain`, вы автоматически тянете много лишнего. Лучше установить точечные пакеты.

Пример полного кода инициализации агента с проверкой версий

Чтобы убедиться, что все работает, создадим скрипт `diagnostics.py`:

```
import langchain
import langchain_openai
import sys

print(f"Python version: {sys.version}")
print(f"LangChain version: {langchain.__version__ if hasattr(langchain, '__version__') else 'Not installed'}")
print(f"LangChain OpenAI version: {langchain_openai.__version__ if hasattr(langchain_openai, '__version__') else 'Not installed'}")

# Проверка ключа
from dotenv import load_dotenv
import os
```

```
load_dotenv()
key = os.getenv("OPENROUTER_API_KEY")
if key:
print("Ключ найден. Длина ключа:", len(key))
print("Первые 5 символов:", key[:5] + "…")
else:
print("Ключ не найден в переменных окружения.")

# Проверка соединения
try:
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(
model="openai/gpt-4o-mini",
openai_api_key=key,
openai_api_base="https://openrouter.ai/api/v1",
max_tokens=10
)
response = llm.invoke("Hi")
print("Соединение успешно! Ответ:", response.content)
except Exception as e:
print("Ошибка соединения:", e)
```

Запустите этот скрипт. Если вы видите "Соединение успешно!", ваше окружение настроено идеально для написания AI-агентов.

Отладка запросов

Если агент ведет себя странно, включите логирование запросов. В Python это можно сделать, установив уровень логирования:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Но будьте осторожны: это залогирует ваш API-ключ в stdout. Используйте это только в безопасной среде.

Также OpenRouter в личном кабинете предоставляет историю запросов, где видно, какой промпт пришел от вашего клиента. Это полезно для отладки форматов сообщений.

Оптимизация стоимости

OpenRouter списывает средства за токены (вход + выход).

1. Используйте дешевые модели для простых задач (GPT-3.5, GPT-4o-mini, Mixtral).

2. Ограничивайте `max_tokens` и `temperature`.

3. Используйте кэширование: если вы часто спрашиваете одно и то же, сохраняйте ответы в Redis или SQLite.

4. Модель "openai/gpt-4o-mini" – оптимальный баланс цены и качества для большинства агентов.

Итог главы

Мы прошли полный цикл настройки: от создания вирту-

ального окружения до написания работающего графа агента с инструментами. Вы научились:

– Безопасно хранить ключи.

– Устанавливать необходимые модули LangChain без лишнего мусора.

– Настраивать ChatOpenAI для работы с OpenRouter.

– Создавать цепочки (Chains) и инструменты (Tools).

– Собирать агента на LangGraph.

В следующей главе мы погрузимся в проектирование агента: как разбить сложную задачу на подзадачи, как настроить память (Memory) для запоминания контекста и как заставить агента планировать свои действия заранее.

Глава 3: Практические реализации AI-агента: от простого чат-бота до автономного исследователя

В этой главе мы наконец отойдем от теории и погрузимся в написание реального кода. Мы последовательно построим три AI-агента разной сложности, используя OpenRouter и LangChain. Каждый следующий проект будет усложнять предыдущий, добавляя новые инструменты и возможности, иллюстрируя, как из простого набора вызовов API вырастает настоящий автономный инструмент. Наша цель – не просто создать работающий пример, но и понять архитектурные решения, почему мы выбираем те или иные компоненты и как с ними правильно работать.

Проект 1: Базовый агент с планированием и экосистемой инструментов

Начнем с классической задачи: создать агента, который способен отвечать на вопросы, требующие актуальных данных, которых нет в базовом обучении языковой модели. В качестве инструментов мы предоставим ему доступ к поиску и вычислительному калькулятору. Этот проект закладывает фундамент для всех последующих: мы научимся инициализировать модель через OpenRouter, определять инструменты, собирать их в цепочку и управлять процессом мышления

агента.

Глава 4: Создание собственных инструментов для агента

В предыдущих главах мы научились создавать цепочки, работать с моделями через OpenRouter и реализовывать базовых агентов, способных использовать предопределенные инструменты. Однако истинная мощь AI-агентов раскрывается только тогда, когда они могут выполнять действия, специфичные для вашего бизнеса или проекта. Стандартные инструменты, такие как поиск в интернете или вычисления, покрывают лишь общие случаи. Чтобы агент стал по-настоящему полезным, ему нужны инструменты, которые знают о вашей базе данных, умеют управлять вашим CRM, развертывать код или анализировать внутренние логи. В этой главе мы подробно разберем, как создавать собственные инструменты (Custom Tools) на Python с использованием LangChain и интегрировать их в агенты, работающие через OpenRouter.

Почему Custom Tools необходимы?

Любой LLM (Large Language Model), работающий через OpenRouter, по своей сути является предсказателем следующего токена. Он не выполняет код и не имеет доступа к внешней среде по умолчанию. Инструменты выступают в качестве "рук" и "ног" модели, позволяя ей взаимодействовать с реальным миром.

Преимущества создания собственных инструментов:

1. Специфичность: Вы можете создать инструмент для работы с уникальным API вашей компании или специфичным файловым форматом.

2. Безопасность: Вы контролируете, какие действия может выполнить агент, ограничивая доступ через API-ключи и логику валидации.

3. Консистентность: Вместо того чтобы заставлять модель генерировать сложные структуры данных (JSON, XML), вы создаете функцию, которая принимает понятные аргументы и возвращает структурированный ответ.

4. Производительность: Инструменты могут кэшировать результаты, выполнять тяжелые вычисления или распараллеливать запросы, недоступные для одной LLM.

Базовая структура инструмента в LangChain

В LangChain инструмент – это класс, который наследуется от базового класса `Tool`. Самый простой способ создать инструмент – использовать декоратор `@tool`. Однако для глубокой интеграции и контроля лучше понимать структуру класса.

Основные компоненты:

– `name`: Уникальное имя инструмента, которое будет видеть LLM.

– `description`: Описание того, что делает инструмент. Это критически важно, так как LLM использует это описание

для выбора нужного инструмента (реакция инструмента).

– `args_schema`: Pydantic-модель, описывающая входные параметры. Это помогает LLM понять, какие аргументы нужно сгенерировать.

– `func`: Асинхронная или синхронная функция, которая выполняет логику инструмента.

Разработка первого инструмента: Простой калькулятор

Давайте начнем с простого примера, чтобы понять механику. Представим, что нам нужен калькулятор, который умеет складывать числа. LLM часто ошибаются в арифметике, поэтому передача вычислений наружу – классический сценарий.

Вариант 1: Использование декоратора `@tool`

```python
from langchain_core.tools import tool
import math

@tool
def calculate_factorial(n: int) -> int:
"""Вычисляет факториал числа n. Используйте для операций с множествами и вероятностями."""
return math.factorial(n)
```

В этом примере LangChain автоматически извлечет имя `calculate_factorial`, описание из строки документации и схе-

му аргументов.

Но для серьезных задач нам нужен полный контроль через класс.

Вариант 2: Кастомный класс инструмента

```python
from langchain_core.tools import BaseTool
from typing import Optional, Type
from pydantic import BaseModel, Field

class CalculatorInput(BaseModel):
"""Схема ввода для калькулятора."""
operation: str = Field(description="Операция: сложение (add), вычитание (sub), умножение (mul) или деление (div)")
a: float = Field(description="Первое число")
b: float = Field(description="Второе число")

class CalculatorTool(BaseTool):
name: str = "calculator"
description: str = "Инструмент для выполнения математических операций. Используйте его для любых вычислений."
args_schema: Type[BaseModel] = CalculatorInput

def _run(self, operation: str, a: float, b: float) -> str:
# Синхронная реализация (для простоты, но в продакшене лучше async)
```

```
try:
if operation == "add":
return str(a + b)
elif operation == "sub":
return str(a – b)
elif operation == "mul":
return str(a * b)
elif operation == "div":
if b == 0:
return "Ошибка: деление на ноль"
return str(a / b)
else:
return "Ошибка: неизвестная операция"
except Exception as e:
return f"Ошибка вычисления: {e}"

# Инициализация
calc_tool = CalculatorTool()
```

Мы создали строго типизированный инструмент. Обратите внимание на `args_schema`. Когда агент получает доступ к этому инструменту, он знает, что нужно передать `operation`, `a` и `b`. Это снижает галлюцинации модели.

Интеграция через OpenRouter и LangChain Agents

Теперь соберем агента, который использует наш каль-

кулятор. Мы будем использовать `create_react_agent` (или `create_tool_calling_agent` в новых версиях LangChain) и подключим модель через OpenRouter.

Предположим, у нас есть промпт:

"Посчитай (5 + 7) * 3 и затем возведи результат в квадрат".

Агент должен разбить задачу: сначала сложить 5+7, потом умножить на 3, потом возвести в квадрат. Или, если модель умная, она попросит калькулятор сделать 12*3 и затем 36^2.

```python
from langchain_openrouter import ChatOpenRouter
from langchain.agents import create_react_agent, AgentExecutor
from langchain_core.prompts import PromptTemplate
import os

# Настройка OpenRouter
# Убедитесь, что у вас установлен пакет: pip install langchain-openrouter
# Для работы через OpenRouter используем стандартный клиент OpenAI, но с другим base_url
from langchain_openai import ChatOpenAI

# Инициализация модели через OpenRouter
# Можно использовать любой доступный через OpenRouter модель, например, Anthropic Claude или GPT-4
```

```
llm = ChatOpenAI(
base_url="https://openrouter.ai/api/v1",
api_key=os.getenv("OPENROUTER_API_KEY"),
model="anthropic/claude-3.5-sonnet", # Пример модели
temperature=0.0
)

# Создаем список инструментов
tools = [CalculatorTool()]

# Шаблон промпта для ReAct агента
prompt_template = """
Отвечай на русском языке. Ты полезный агент, который может использовать инструменты.
Инструменты доступны: {tool_names}

Инструкции:
1. Проверь, нужно ли использовать инструмент для ответа на вопрос.
2. Если да, сгенерируй вызов инструмента в формате:
Thought: [Твои рассуждения]
Action: [Имя инструмента]
Action Input: [Входные данные]
Observation: [Результат выполнения]
3. Повторяй шаги 1-3, пока не получишь окончательный ответ.
```

4. Не придумывай результаты инструментов.

Вопрос: {input}
{agent_scratchpad}
"""

prompt = PromptTemplate.from_template(prompt_template)

# Создаем агента
agent = create_react_agent(llm, tools, prompt)

# Создаем исполнитель агента
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# Запуск
question = "Посчитай (5 + 7) * 3 и затем возведи результат в квадрат."
response = agent_executor.invoke({"input": question})
print(response['output'])
```

В этом коде `create_react_agent` подготавливает промпт, вставляя описание инструментов. LLM через OpenRouter получает вопрос, планирует использование инструмента и
```

генерирует структуру вызова. LangChain парсит этот вызов, выполняет `_run` нашего `CalculatorTool` и возвращает результат обратно модели для продолжения рассуждения.

Создание асинхронных инструментов для производительности

В веб-приложениях и высоконагруженных системах синхронные инструменты блокируют выполнение. LangChain полностью поддерживает асинхронность. Это критично, если ваш инструмент делает запросы к базе данных или внешним API.

Для этого нужно:

1. Унаследоваться от `BaseTool`.

2. Реализовать метод `_arun` (асинхронный аналог `_run`).

3. Если `_arun` не реализован, LangChain попытается запустить `_run` в `run_in_executor`, но лучше писать нативный асинхронный код.

Пример инструмента для запроса к внешнему API (Асинхронный):

```python
import aiohttp
from langchain_core.tools import BaseTool
from pydantic import BaseModel, Field

class CryptoPriceInput(BaseModel):
```

```
    coin_id: str = Field(description="ID криптовалюты (напри-
мер, bitcoin, ethereum)")

    class CryptoPriceTool(BaseTool):
    name: str = "get_crypto_price"
    description: str = "Получает текущую цену криптовалюты
в USD. Используй для финансовых вопросов."
    args_schema: Type[BaseModel] = CryptoPriceInput

    async def _arun(self, coin_id: str) -> str:
    # Асинхронный запрос к CoinGecko API (или любому
другому)
    url = f"https://api.coingecko.com/api/v3/simple/price?
ids={coin_id}&vs_currencies=usd"
    try:
    async with aiohttp.ClientSession() as session:
    async with session.get(url) as response:
    if response.status == 200:
    data = await response.json()
    if coin_id in data:
    price = data[coin_id]['usd']
    return f"Цена {coin_id} сейчас: ${price}"
    return "Монета не найдена"
    return f"Ошибка API: {response.status}"
    except Exception as e:
    return str(e)
```

```
    def _run(self, coin_id: str) -> str:
    # Заглушка для синхронного режима (или можно вызвать синхронный requests)
    # В реальном проекте лучше вызывать асинхронный код через asyncio.run
    import asyncio
    return asyncio.run(self._arun(coin_id))

    crypto_tool = CryptoPriceTool()
```

Обратите внимание, что в `_arun` мы используем `aiohttp`. Теперь, если агент будет запущен в асинхронном режиме (например, внутри FastAPI), этот инструмент не будет блокировать поток.

Интеграция с базами данных (SQL Tool)

Один из самых популярных сценариев – позволить агенту общаться с базой данных. В LangChain есть готовые инструменты (`SQLDatabaseToolkit`), но создание своего кастомного SQL-инструмента дает больше контроля над безопасностью.

Допустим, у нас есть SQLite база данных с таблицей `products`.

Нам нужен инструмент, который выполняет запрос и возвращает результат. Чтобы избежать SQL-инъекций, мы не

будем передавать сырой SQL от LLM. Вместо этого мы напишем функцию, которая выполняет селект по названию продукта.

```python
from langchain_core.tools import BaseTool
from pydantic import BaseModel, Field
import sqlite3

class ProductSearchInput(BaseModel):
product_name: str = Field(description="Название продукта для поиска")

class ProductSearchTool(BaseTool):
name: str = "sql_product_search"
description: str = "Ищет информацию о продукте в базе данных по его названию. Возвращает цену и описание."
args_schema: Type[BaseModel] = ProductSearchInput

def _run(self, product_name: str) -> str:
# В реальном продакшене используйте пулы соединений (например, SQLAlchemy)
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

Глава 5. Агент с долгосрочной памятью на базе чата

Создание агентов, способных поддерживать непрерывный, осмысленный диалог с пользователем, является одной из ключевых задач при разработке современных интерактивных систем. Обычные чат-боты, работающие в режиме «запрос-ответ», не учитывают предыдущий контекст, что делает взаимодействие фрагментарным и неестественным. Для решения этой проблемы в LangChain реализован механизм, объединяющий под управлением агента компоненты памяти, планирования и выполнения действий. Разделение логики позволяет агенту анализировать историю диалога, извлекать из неё релевантные факты и строить стратегию дальнейшего взаимодействия. Агент с долгосрочной памятью на базе чата (Long-Term Memory Chat Agent) – это архитектура, в которой Large Language Model (LLM) выступает в роли ядра мышления, а специальные хранилища данных (векторные базы или базы ключ-значение) обеспечивают доступ к накопленным знаниям. В этой главе мы детально разберем этапы проектирования, реализации и отладки такого агента с использованием openrouter.ai и LangChain.

Предварительные требования и настройка окружения

Перед началом работы необходимо подготовить инфра-

структуру. Поскольку агент будет взаимодействовать с внешними API и базами данных, важно корректно настроить переменные окружения и установить необходимые библиотеки. Основной язык программирования – Python (версия 3.9 и выше). Для работы с LLM через openrouter.ai мы будем использовать стандартные HTTP-клиенты или интеграции LangChain, для работы с памятью – специализированные модули, для хранения эмбеддингов – библиотеку Chromadb или FAISS.

Установка зависимостей:

pip install langchain langchain-openai chromadb faiss-cpu openai python-dotenv

Ключевые компоненты архитектуры

Любой агент в LangChain состоит из трех основных блоков:

1. Модель (LLM): «Мозг» агента. Мы будем использовать модели, доступные через openrouter.ai. Это позволяет выбирать самые мощные модели (например, GPT-4, Claude, Llama) без привязки к одному провайдеру.

2. Память (Memory): Хранилище состояния. Она делится на краткосрочную (контекст текущего окна диалога) и долгосрочную (архив знаний и фактов).

3. Инструменты (Tools): Специальные функции, расширяющие возможности агента (поиск информации, выполнение расчетов, доступ к API).

Наша задача – связать эти компоненты так, чтобы агент

мог «вспоминать» прошлые взаимодействия, даже если они были давно, и использовать эти воспоминания для текущих ответов.

Подключение модели через OpenRouter

OpenRouter предоставляет единый API для множества моделей. Для работы в LangChain нам требуется создать клиент OpenAI-совместимого формата, указав базовый URL OpenRouter.

Пример конфигурации модели:

```
import os
from langchain_openai import OpenAI
from dotenv import load_dotenv
load_dotenv()
# Указываем URL API OpenRouter вместо стандартного OpenAI
OPENROUTER_API_KEY = os.getenv("OPENROUTER_API_KEY")
BASE_URL = "https://openrouter.ai/api/v1"
llm = OpenAI(
base_url=BASE_URL,
api_key=OPENROUTER_API_KEY,
model="openai/gpt-3.5-turbo", # Можно выбрать любую доступную модель
temperature=0.7
)
```

Важно помнить, что при использовании агентов с па-

мятью, особенно с длинной историей, стоимость токенов может расти. Поэтому настройка параметров (temperature, max_tokens) критична для оптимизации.

Краткосрочная память как основа диалога

Прежде чем переходить к «архиву» знаний, агент должен помнить текущую беседу. В LangChain для этого используется класс ConversationBufferMemory. Он хранит всю историю сообщений в виде списка, который передается в LLM как часть промпта. Это базовый слой, необходимый для любого чат-агента.

Однако для полноценного агента нам нужно сохранять не только фразы, но и контекст выполнения. Поэтому мы используем ConversationBufferMemory в связке с версией для агентов.

Пример инициализации памяти:

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
```

Память должна быть доступна агенту, но передаваться она должна в специальном формате, который агент понимает как историю диалога.

Долгосрочная память: Vector Store и Retrieval

Самая интересная часть – реализация долгосрочной памяти. Хранить весь текст диалога в контексте модели невозможно из-за ограничений длины контекстного окна (context

window). Решение – хранить информацию во внешней базе и извлекать её только тогда, когда это необходимо.

Для этого используется механизм Retrieval-Augmented Generation (RAG). Идея следующая:

1. Когда пользователь сообщает важный факт (например, «Я живу в Новосибирске»), агент парсит эту информацию и сохраняет её в векторную базу данных.

2. При последующих вопросах (например, «Какая погода в моем городе?») агент ищет в базе релевантные записи по семантическому сходству (векторный поиск).

3. Найденная информация подставляется в контекст запроса к LLM.

Для реализации нам понадобятся:

Векторная база (Chroma).

Функция вложения (Embeddings). Мы будем использовать встроенные эмбеддинги OpenAI (или доступные через OpenRouter, но обычно для эмбеддингов используют отдельные модели, например, text-embedding-ada-002).

Пример настройки векторного хранилища:

```
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings
# Инициализация эмбеддингов
embeddings = OpenAIEmbeddings(
base_url=BASE_URL,
api_key=OPENROUTER_API_KEY,
model="text-embedding-ada-002" # Эта модель обычно до-
```

ступна через OpenRouter или OpenAI
)
# Создание или загрузка базы
vectorstore = Chroma(
collection_name="agent_memory",
embedding_function=embeddings,
persist_directory="./chroma_db"
)

Чтобы агент мог использовать эту базу как память, нам нужен специальный инструмент (Tool), который будет выполнять операции сохранения и поиска. Но сначала нужно определить, как именно агент решает, когда сохранять информацию.

Стратегия сохранения информации (Memory Management)

Агент не должен сохранять каждую фразу подряд, иначе база заполнится мусором. Нам нужен механизм принятия решения о важности информации. В LangChain это реализуется через концепцию «Планировщика» (Planner) или нативно через промпт агента.

Мы можем реализовать простую стратегию:

1. Использовать LLM для классификации текущего сообщения пользователя.

2. Если сообщение содержит факты (имя, место, предпочтения), LLM форматирует его в структурированную запись (JSON) и отправляет в инструмент сохранения.

3. Если сообщение – просто реплика в диалоге, оно идет только в краткосрочную память (контекст).

Для реализации этой логики создадим кастомный инструмент (Custom Tool).

Кастомный инструмент долгосрочной памяти

Класс инструмента в LangChain наследуется от BaseTool. Нам нужно два инструмента:

1. `SaveMemory`: сохраняет контекст в Chroma.

2. `LoadMemory`: извлекает контекст по запросу.

Реализация инструмента сохранения:

```
from langchain.tools import BaseTool
from typing import Optional, Type
from pydantic import BaseModel, Field
class MemoryInput(BaseModel):
content: str = Field(description="Содержание для сохранения в долговременную память")
category: Optional[str] = Field(default="general", description="Категория воспоминания")

class SaveMemoryTool(BaseTool):
name = "save_memory"
description = "Используй этот инструмент, чтобы сохранить важную информацию о пользователе или факты для долгосрочного использования. Содержание должно быть кратким и информативным."
args_schema: Type[BaseModel] = MemoryInput
```

```
    def _run(self, content: str, category: str = "general") -> str:
    # Сохранение в векторную базу
    try:
    vectorstore.add_texts(
    texts=[content],
    metadatas=[{"category": category}]
    )
    return "Информация успешно сохранена в памяти."
    except Exception as e:
    return f"Ошибка при сохранении: {e}"
```

Нам также понадобится инструмент для поиска. Однако, чтобы не перегружать контекст, мы можем сделать «умный» поиск: агент передает запрос, а инструмент возвращает релевантные фрагменты.

```
    class SearchInput(BaseModel):
    query: str = Field(description="Поисковый запрос для извлечения воспоминаний")


    class LoadMemoryTool(BaseTool):
    name = "retrieve_memory"
    description = "Используй этот инструмент, чтобы найти сохраненную информацию о пользователе или прошлых событиях, если это необходимо для ответа. Запрос должен содержать ключевые слова или суть того, что вы ищете."
    args_schema: Type[BaseModel] = SearchInput
    def _run(self, query: str) -> str:
```

try:
docs = vectorstore.similarity_search(query, k=3)
if not docs:
return "Нет релевантных воспоминаний."
# Форматируем результаты
result_text = "\n".join([doc.page_content for doc in docs])
return f"Найденные воспоминания:\n{result_text}"
except Exception as e:
return f"Ошибка при поиске: {e}"
Сборка агента: Планирование и Очередь

Теперь, когда у нас есть инструменты памяти, краткосрочная память и модель, мы можем собрать агента. Но есть нюанс. Стандартный агент LangChain (AgentExecutor) выполняет инструменты последовательно, пока не получит финальный ответ. Нам же нужна более сложная логика:

1. Анализ текущего ввода.

2. Решение: нужно ли сохранять информацию прямо сейчас? (Это можно делать как отдельный шаг агента, так и через промпт).

3. Решение: нужно ли извлекать информацию из долгосрочной памяти?

Для упрощения мы можем использовать `AgentExecutor` с системой промптов, которая заставляет агента думать о памяти.

Однако более эффективный подход для "сценария чата" – использование `ConversationalAgent` (или

`ToolCallingAgent`) с настроенной памятью.

Мы будем использовать `initialize_agent` из LangChain.

Важно: Агенту нужно передать инструменты, LLM, память и промпт.

Однако стандартная память `ConversationBufferMemory` в `initialize_agent` работает только как история сообщений. Чтобы встроить векторную базу, мы должны немного изменить подход.

Вместо того чтобы заставлять агента постоянно вызывать инструменты поиска, мы можем использовать подход `RetrievalQA` внутри логики агента, или сделать так, чтобы агент сам вызывал инструменты.

В нашем случае мы сделаем так: агент видит инструменты `SaveMemoryTool` и `LoadMemoryTool`. В промпте мы четко пропишем, когда их использовать.

Но для чистоты эксперимента и чтобы избежать бесконечных циклов вызова инструментов, лучше использовать `create_react_agent` (или `create_tool_calling_agent`), где инструменты памяти будут доступны, но агент будет использовать их разумно.

Промпт агента

Ключевой элемент – промпт. Он должен содержать инструкции:

* Ты – полезный ассистент с долгосрочной памятью.

* Ты можешь сохранять факты о пользователе, используя инструмент `save_memory`.

* Если пользователь спрашивает о том, что могло быть сказано ранее, используй `retrieve_memory`.

* История диалога доступна в переменной {chat_history}.

* {agent_scratchpad} – место для мыслей и инструментов.

Создадим промпт:

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain.agents import AgentExecutor, create_tool_calling_agent
# Базовый системный промпт
system_prompt = """
```

Вы – полезный ИИ-ассистент с расширенной долгосрочной памятью.

Ваши возможности:

1. Вы должны поддерживать непрерывный диалог.

2. Если пользователь делится личной информацией (имя, предпочтения, факты о жизни), вы ОБЯЗАНЫ использовать инструмент `save_memory` для сохранения этой информации.

3. Если пользователь задает вопрос, который требует знания фактов из прошлого (которые могли быть сказаны давно или в другом контексте), вы ОБЯЗАНЫ сначала использовать `retrieve_memory` для поиска релевантной информации.

4. Не изобретайте факты, опирайтесь только на историю диалога и результаты из памяти.

5. Всегда отвечай на русском языке.

"""
# Шаблон для агента (Tool Calling Agent)
prompt = ChatPromptTemplate.from_messages([
("system", system_prompt),
MessagesPlaceholder("chat_history"),
("user", "{input}"),
MessagesPlaceholder("agent_scratchpad"),
])
Инициализация агента
Теперь собираем всё вместе.
from langchain.agents import AgentExecutor
from langchain_openai import ChatOpenAI # Используем ChatOpenAI для поддержки функций/инструментов
# Используем ChatOpenAI для агента, так как он лучше работает с инструментами
# Но можно оставить и OpenAI, если переделать под существующие агенты.
# Для надежности используем ChatOpenAI через OpenRouter
llm_chat = ChatOpenAI(
base_url=BASE_URL,
api_key=OPENROUTER_API_KEY,
model="openai/gpt-3.5-turbo", # Модель с поддержкой function calling
temperature=0.3
)

```
    # Наши инструменты
    tools = [SaveMemoryTool(), LoadMemoryTool()]
    # Создаем агента
    agent = create_tool_calling_agent(llm_chat, tools, prompt)
    # Исполнитель агента
    agent_executor = AgentExecutor(agent=agent, tools=tools,
verbose=True, handle_parsing_errors=True)
    # Инициализация памяти (для хранения истории чата)
    chat_memory =
ConversationBufferMemory(memory_key="chat_history",
return_messages=True)
    # Тестовый цикл взаимодействия
    def run_chat():
    print("Агент запущен. Введите 'exit' для выхода.")
    while True:
    user_input = input("Вы: ")
    if user_input.lower() == 'exit':
    break

    # Получаем историю из памяти
    history = chat_memory.load_memory_variables({})
    chat_history = history.get("chat_history", [])

    # Запуск агента
    try:
    response = agent_executor.invoke({
```

```
    "input": user_input,
    "chat_history": chat_history
    })
    answer = response['output']
    print(f"Агент: {answer}")

    # Сохраняем оба сообщения в память (историю диалога)
    chat_memory.save_context({"input": user_input}, {"output":
answer})
    except Exception as e:
    print(f"Ошибка: {e}")

    if __name__ == "__main__":
    run_chat()
```

Детальное объяснение работы цикла:

1. Пользователь вводит сообщение.
2. Мы загружаем историю диалога из `ConversationBufferMemory`.
3. Передаем в `agent_executor`: ввод, историю и доступные инструменты.
4. Агент анализирует запрос. Если он решает, что нужно сохранить информацию, он вызывает `SaveMemoryTool`. Мы видим это в `verbose` режиме. Этот вызов сохраняет данные в Chroma.
5. Если он решает, что нужно вспомнить прошлое, он вызывает `LoadMemoryTool`. Данные из Chroma возвращаются

в контекст.

6. Агент генерирует финальный ответ на основе текущего ввода, истории и полученных из памяти фактов.

7. Ответ выводится пользователю и сохраняется в `ConversationBufferMemory`.

Продвинутое использование: Инструмент с реверсивной записью

Вышеприведенный код работает, но у него есть недостаток: агент должен явно вызывать инструменты `save_memory` и `retrieve_memory` через шаги мышления (Thought -> Action -> Observation). Это замедляет общение. Мы можем автоматизировать этот процесс, используя концепцию «Трансформеров памяти» (Memory Transformers) или «Скрытых действий» (Hidden Actions).

В LangChain есть возможность создать инструмент, который вызывается автоматически как часть промпта (через скрытый мыслительный процесс), но для простоты и прозрачности мы оставим явный вызов.

Однако мы можем улучшить инструмент `SaveMemoryTool`. Вместо простого сохранения строки, он может принимать JSON-структуру, чтобы обогатить метаданные.

Улучшенный инструмент сохранения:

class EnhancedSaveMemoryTool(BaseTool):

name = "save_memory_enhanced"

description = "Сохраняет информацию в долговременную память. Используй этот инструмент для сохранения фактов, имен, мест, предпочтений пользователя. Сохраняй только суть, убирая лишние слова."
args_schema: Type[BaseModel] = MemoryInput
def _run(self, content: str, category: str = "general") -> str:
# Здесь можно добавить векторизацию и сохранение
try:
# Допустим, мы хотим сохранять исходный контекст
vectorstore.add_texts(
texts=[f"Факт: {content}"],
metadatas=[{"source": "long_term_memory", "category": category}]
)
return "Факт сохранен."
except Exception as e:
return f"Ошибка сохранения: {e}"
Решение проблем и отладка

При работе с агентами памяти часто возникают следующие проблемы:

1. **Забывание контекста инструмента**: Агент вызывает инструмент, получает ответ (например, "Факт сохранен"), но забывает об этом в генерации финального ответа.

* *Решение*: В `agent_scratchpad` (истории мыслей) сохраняются все шаги. Модель видит, что инструмент был вызван, и обычно учитывает это. Если нет – можно добавить

инструкцию в промпт: "После выполнения инструмента сделай вывод на основе Observation".

Конец ознакомительного фрагмента.