



Инженер

LangChain и OpenSCAD: Практическая реализация AI-агента

Инженер

**LangChain и OpenSCAD:
Практическая
реализация AI-агента**

«Автор»

2026

Инженер

LangChain и OpenSCAD: Практическая реализация AI-агента /
Инженер — «Автор», 2026

Книга — практическое руководство по созданию интеллектуальных AI-агентов с интеграцией двух технологий: LangChain (для проектирования цепочек рассуждений, работы с LLM и внешними данными); OpenSCAD (для генерации 3D-моделей и визуализации данных через параметрическое проектирование). Вы научитесь: настраивать окружение и связывать LangChain с OpenSCAD; создавать агентов, способных анализировать задачи и выдавать 3D-решения; проектировать цепочки: от текстового запроса → к коду OpenSCAD → к готовой модели; автоматизировать генерацию прототипов, технических чертежей и визуализаций. Практические кейсы: агент для конвертации описаний в 3D-объекты; система проектирования простых механизмов по текстовому ТЗ; визуализация данных (графики, схемы) в 3D. Для кого книга: разработчики AI-приложений; инженеры и дизайнеры, внедряющие AI в проектирование; исследователи, сочетающие NLP и 3D-моделирование. Результат: вы сможете создавать агентов, превращающих текст в работающие 3D-модели.

© Инженер, 2026

© Автор, 2026

Содержание

Глава 1. Введение в LangChain и OpenSCAD	5
Глава 2. Основы построения AI-метапрограмм	8
Конец ознакомительного фрагмента.	13

Инженер LangChain и OpenSCAD: Практическая реализация AI-агента

Глава 1. Введение в LangChain и OpenSCAD

Введение в LangChain и OpenSCAD и практический сценарий создания AI-агента, способного генерировать модели параметрических деталей с помощью комбинации мощного фреймворка LangChain и детализированного язык-программирования OpenSCAD может стать отправной точкой для построения целых экосистем автоматического проектирования; в этом разделе мы подробно разберём архитектуру LangChain, принципы работы его компоненто-в-агентов, способы интеграции с OpenSCAD через скриптовые вызовы и демонстрируем, как собрать полностью рабочий прототип агента, способного принимать естественную речь пользователя, интерпретировать запросы о форме, размерах и функциональных характеристиках, а затем преобразовывать их в готовый код OpenSCAD и отправлять готовый STL-файл на виртуальный или реальный 3-D-принтер; сначала необходимо понять, что LangChain представляет собой набор модулей и цепочек (chains), позволяющих связывать между собой вызовы LLM, базы данных, внешних API и пользовательских функций так, чтобы построить сложные сценарии обработки информации, при этом каждый компонент (prompt, model, memory, tool, output parser) играет определённую роль в минимизации ложных интерпретаций и повышении воспроизводимости результатов; в контексте OpenSCAD агент будет оперировать не только текстовыми данными, но и структурированными процедурными скриптами, поэтому важно обеспечить методы передачи кода в обе стороны: от LLM к OpenSCAD (выполнение внешних процессов) и от OpenSCAD к LLM (получение обратной связи о структуре, визуализации модели и возможности её доработки); один из самых простых и одновременно гибких подходов подразумевает создание собственного `Tool`-класса, наследующего базовый интерфейс LangChain, который будет отвечать за запуск внешних команд `openscad` из подпроцесса, чтение результата их выполнения (например, выдаваемый G-code или файл `.scad`) и возврат его содержимого в виде строки, которую последующий `output parser` сможет интерпретировать как готовый Beschreibung или как набор параметров, подлежащих повторному запросу у модели; благодаря такой интеграции агент получает возможность выполнять не только генерацию чисто текстовых ответов, но и реального производства 3-D-моделей, что открывает путь к автоматизированной конвейерной обработке запросов от конечных пользователей, работающих в CAD-системах, инженерных отделах или сервисах быстрого прототипирования; следующий шаг состоит в построении цепочки (chain) LangChain, в которой запрос пользователя сначала обрабатывается prompts-модулем, который формирует ввод-текст для LLM, затем запрос передаётся через `LLMChain`, где модель генерирует промпт для получения спецификации модели (размер, шаги, материалы, особенности), далее этот промпт отправляется в наш кастомный `OpenSCADExecutorTool`, который создаёт временный файл `.scad` со всеми вычисленными параметрами и вызывает OpenSCAD через `subprocess` для компиляции модели; после генерации файл передаётся в `STLExportStep` – встроенную функцию, которая использует OpenSCAD в режиме headless (`openscad -o output.stl model.scad`) и в результате получает готовый STL-файл, который затем может быть сохранён, отправлен в очередь печати или возвращён пользователю в виде ссылки на файл; каждый из этих шагов можно снабдить механизмами памяти (`memory`), так что агент будет помнить предыдущие запросы,

поддерживать контекст и корректировать последующие генерации, тем самым обеспечивая более персонализированный опыт и сокращая количество повторных уточнений; для реализации такой цепочки в коде на Python потребуется установить несколько библиотек: `langchain`, `langchain\_community` (для новых модулей), `openai` или любой другой провайдер LLM-API, а также `subprocess`, `os`, `tempfile` и `re` для работы с внешними процессами и регулярными выражениями; пример инициализации агента выглядит следующим образом:

```
`from langchain.agents import Tool, AgentType`
`from langchain.llms import OpenAI`
`from langchain.chains import LLMChain`
`from langchain.prompts import PromptTemplate`
`import subprocess, tempfile, os, re`
`llm = OpenAI(model="gpt-4-turbo", temperature=0.2)`
`scad_prompt = PromptTemplate(input_variables=["spec"], template="Generate a complete`
OpenSCAD script based on the following specification: {spec}")`
`spec_chain = LLMChain(llm=llm, prompt=scad_prompt)`
`def run_scad(script):`
`    with tempfile.NamedTemporaryFile(mode='w', suffix='.scad', delete=False) as f:`
`        f.write(script)`
`        script_path = f.name`
`        stl_path = script_path.replace('.scad', '.stl')`
`        subprocess.run(['openscad', '-o', stl_path, script_path], check=True)`
`        return stl_path`
`scad_tool = Tool(name="OpenSCADExecutor", func=run_scad, description="Executes an`
OpenSCAD script and returns the path to the generated STL file.")`
`agent = AgentType.create(agent_name="OpenSCADAgent", tools=[scad_tool])`
```

Тут `AgentType.create` – гипотетический метод, позволяющий собрать агента из набора инструментов; в реальной реализации обычно используют `initialize\_agent` или `AgentExecutor` из более новых версий библиотеки; важно отметить, что `AgentExecutor` автоматически управляет промпт-ами, мемогу и выводом, поэтому он может заменить ручное построение цепочки; при этом каждый вызов `execute` будет проверять, требуется ли запрос к инструменту `OpenSCADExecutor`, и автоматически преобразовывать полученный путь к STL-файлу в ответ пользователю или в сообщение об ошибке, если скрипт не скомпилировался; следующий пример демонстрирует полный рабочий цикл:

```
`def ask_user(request_text):`
`    # 1. Сформировать промпт для LLM`
`    spec = spec_chain.run(request_text)`
`    # 2. Передать скрипт в OpenSCADExecutorTool`
`    stl_path = scad_tool.run(spec)`
`    # 3. Вернуть путь к файлу`
`    return stl_path`
```

В этом сценарии пользователь отправляет запрос через любой канал – чат-бота, веб-интерфейса или телефонную линию – а агент, используя `AgentExecutor`, последовательно обрабатывает запрос, генерирует спецификацию, компилирует её в OpenSCAD и возвращает готовый STL-файл; при желании можно добавить дополнительный слой – `output parser`, который будет трансформировать путь к STL в URL, который пользователь сможет

открыть непосредственно в браузере; также можно внедрить механизм ``memory`` (например, ``ConversationTokenMemory``), чтобы агент помнил предыдущие обсуждения о ранге материалов, размерах печати или требованиях к поверхности, тем самым уменьшает количество корректировок и делает генерацию более предсказуемой; в процессе тестирования пользователь может вводить уточнения вроде ``"увеличьте радиус в 1.5 раза"`` или ``"добавьте отверстие диаметром 5 мм в центре"``, которые агент интерпретирует заново, либо сохраняет в контексте для последующего предиката, тем самым создавая гибридный диалог, в котором естественный язык и технические требования плавно переходят друг в друга; при масштабировании такой системы на множество типов моделей (например, сложные механизмы, артефактные геометрии или многочастичные сборки) можно построить набор специализированных инструментов: ``ExtrudeTool``, ``RotateTool``, ``LoopTool``, каждый из которых реализует специфичную операцию над уже сгенерированным скриптом и может быть вызван последовательно в цепочке; при этом ``Agent`` будет автоматически определять, какой инструмент нужен на основе анализа промпта, и выполнять соответствующую функцию, позволяя пользователю описывать модель почти свободным языком без необходимости писать реальный код OpenSCAD; такой подход открывает значительные возможности для дальнейшего развития: от интеграции с облачными сервисами хранения STL-файлов, через автоматический импорт готовых библиотек библиотек OpenSCAD (``<mcad>``, ``<cga>`` и др.) до построения полноценных CI/CD пайплайнов, где изменения в спецификации автоматически порождают новые версии моделей, проходят тесты на валидность стенок и отправляются в очередь печати без человеческого вмешательства; в конечном итоге практическая реализация агента на базе LangChain и OpenSCAD демонстрирует, как современные LLM-модели могут стать «мостом» между свободной коммуникацией и детализированным механическим проектированием, позволяя как новичкам без глубоких знаний CAD-техники быстро создавать прототипы, так и инженерам ускорять итеративный процесс разработки за счёт автоматизированного рендеринга и экспорта; для дальнейшего развития рекомендуется протестировать разные модели LLM – от GPT-3.5-Turbo до специализированных моделей, оптимизированных под генерацию технических описаний, исследовать варианты «chain-of-thought» для более точного парсинга пользовательских запросов, а также добавить слои валидации, такие как проверка единиц измерения, диапазонов размеров и корректности синтаксиса OpenSCAD, чтобы гарантировать, что каждая сгенерированная модель будет пригодна к печати без необходимости ручной правки; таким образом, глава «Введение в LangChain и OpenSCAD» дает полное представление о том, как построить многофункционального AI-агента, способного принимать естественную речь, преобразовывать её в детализированную параметрическую модель в OpenSCAD, вычислять её, экспортировать STL и доставлять готовый объект на 3-D-принтер, при этом используя гибкую архитектуру LangChain, промпты, память и кастомные инструменты, что делает систему масштабируемой, пользовательско-ориентированной и готовой к интеграции в реальные промышленные и исследовательские проекты.

Глава 2. Основы построения AI-метапрограмм

Введение в создание AI-агента в OpenSCAD через LangChain

OpenSCAD — это скриптовый язык моделирования твердых тел, широко используемый в техническом проектировании и 3D-печати. Нативное API OpenSCAD ориентировано на декларативное описание геометрии, но для взаимодействия с внешними данными, динамического выбора параметров и генерации сложных моделей часто требуется более гибкое управление. Именно здесь вступает в игру LangChain, позволяющая построить так называемый AI-агент, который может автоматически генерировать код OpenSCAD, учитывая запросы пользователя, ограничения проектирования и результаты предыдущих шагов.

В этом разaking мы рассмотрим пошаговый процесс написания такого агента, опираясь на возможности библиотеки LangChain, а также покажем практико-ориентированный формат кода, который можно сразу использовать в своих проектах.

1. Подготовка окружения

Для начала необходимо установить необходимые зависимости. На Python-окружении рекомендуется создать виртуальное окружение и установить пакеты langchain-core, langchain-openai (или другой провайдер LLM-модели), а также библиотеку python-openSCAD, которая обеспечивает возможность генерировать и отправлять команды OpenSCAD из кода Python. Кроме того, потребуется доступ к LLM-модели, например, GPT-4 или аналог.

Команды установки выглядят так:

```
python -m venv venv
source venv/bin/activate # Linux/macOS
venv\Scripts\activate # Windows
pip install langchain-core langchain-openai python-openSCAD
```

После установки необходимо задать переменные окружения для доступа к LLM: OPENAI_API_KEY и другие параметры, связанные с выбранным провайдером.

2. Общая архитектура агента

AI-агент в контексте LangChain представляет собой цепочку (pipeline) of LLM-вызовов, правил и вспомогательных компонентов. Для OpenSCAD-сценария типичная структура выглядит так:

- Входные данные — запрос пользователя, описывающий требуемую модель (например, «создай случайную спираль с радиусом от 5 до 15 мм и пяти estivessetami»).
- Преобразователь запроса (prompt template) — переводит естественный язык в промпт, пригодный LLM.
- LLM-вызов — запрашивает у модели генерацию кода OpenSCAD.
- Пост-обработка — фильтрует полученный код: проверка синтаксиса, корректировка параметров, приведение к формату, совместимому с OpenSCAD.

- Вывод – передача готового кода в OpenSCAD или сохранение в файл.

LangChain предоставляет готовые классы для построения таких цепочек: PromptTemplate, LLMChain, SequentialChain и др. В нашем случае наиболее подходящим вариантом будет использование LLMChain с пользовательским обработчиком вывода (output parser), который преобразует строку кода в объект, который можно непосредственно выполнить.

3. Создание шаблона промпта

Промпт должен четко формулировать требования к коду: указать типы геометрических примитивов, их параметры, отношения между элементами, а также ограничения по размеру модели. Пример шаблона:

Ты – эксперт по OpenSCAD. Сгенерируй только корректный код OpenSCAD, соответствующий следующему описанию:

{user_request}

Обязательно включи модуль main() и заверши код оператором render()!\nКод:

Здесь {user_request} будет заменяться на реальный запрос пользователя. Подобный шаблон гарантирует, что модель будет выводить именно код, а не объяснительные тексты.

4. Настройка LLMChain

С помощью LLMChain мы свяжем промпт, модель и обработчик. Пример реализации:

```
from langchain import LLMChain
from langchain.prompts import PromptTemplate
from langchain_openai import ChatOpenAI
```

```
llm = ChatOpenAI(model="gpt-4", temperature=0.2, max_tokens=1500)
```

```
prompt = PromptTemplate(
    input_variables=["user_request"],
    template=(
        "Ты – эксперт по OpenSCAD. Сгенерируй только корректный код OpenSCAD, "
        "соответствующий описанию:\n{user_request}\n"
        "Обязательно включи модуль main() и заверши код оператором render()!\n"
        "Код:"
    )
)
```

```
chain = LLMChain(
    llm=llm,
    prompt=prompt,
    output_parser=OpenSCADParser()
```

```
)  
'''
```

В данном примере `OpenSCADParser` – пользовательский класс, реализующий парсинг полученного текста в валидный набор команд. Он будет подробно рассмотрен ниже.

5. Реализация парсера кода OpenSCAD

Получаемый от LLM текст может содержать лишние префиксы, пояснительные строки и даже ошибки синтаксиса. Чтобы гарантировать корректную работу, нужен парсер, который:

- Отрезает все строки до первого появления слова `module` или `//=== START CODE ===//`.
- Удаляет комментарии в конце строк.
- Проверяет наличие обязательных блоков `module main()` и `render()`; при их отсутствии добавляет.
- Возвращает чистый код в виде строки.

Пример реализации простого парсера:

```
'''
```

```
import re
```

```
class OpenSCADParser:  
    def parse(self, text: str) -> str:  
        # Удаляем всё до первого признака начала кода  
        match = re.search(r"(?s)(module\s+\w+|!?\s*(\s*?)//=== START CODE ===//)", text)  
        if not match:  
            raise ValueError("Не найден маркер начала кода")  
        start_idx = match.start(1) if match.lastgroup == "module" else match.start("START CODE")  
        code = text[start_idx:].strip()  
        # Удаляем комментарии в конце строк  
        lines = code.splitlines()  
        processed = []  
        for line in lines:  
            if "#" in line:  
                line = line[: line.index("#")].rstrip()  
            processed.append(line)  
        code = "\n".join(processed)  
        # Убеждаемся, что есть модуль main и render  
        if "module main()" not in code:  
            code += "\nmodule main() {\n}"  
        if "render()" not in code:  
            code += "\nrender()\n}"  
        # Финальная очистка лишних пустых строк  
        code = re.sub(r"\n{3,}", "\n\n", code).strip()  
        return code  
'''
```

Этот парсер достаточно прост, но в реальных проектах можно расширить его, добавив проверку соответствия синтаксису OpenSCAD через подпроцесс `openscad -c` и откат к повторному запросу модели при ошибке.

6. Интеграция с OpenSCAD

После получения чистого кода его можно выполнить в OpenSCAD через командную строку или через Python-интерфейс. Если требуется дальнейшая обработка (например, экспорт STL), удобно создать функцию `execute\_openscad(code: str) -> str`, которая:

1. Записывает код в временный файл `temp.scad`.
2. Вызывает OpenSCAD в режиме командной строки: `openscad -c temp.scad -o temp.stl`.
3. Возвращает путь к полученному STL-файлу или содержимое, если нужен только 2D-проект.

Пример реализации:

```
'''
import os
import tempfile
import subprocess

def execute_openscad(code: str) -> str:
    # Создаём временный файл
    with tempfile.NamedTemporaryFile(mode="w", suffix=".scad", delete=False) as f:
        f.write(code)
    temp_path = f.name
    try:
        # Путь к исполняемому файлу OpenSCAD
        openscad_path = "openscad" # Предполагаем, что он доступен в PATH
        # Формируем команду для генерации STL
        cmd = [openscad_path, "-c", temp_path, "-o", temp_path.replace(".scad", ".stl")]
        subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL,
stderr=subprocess.DEVNULL)
        # Возвращаем путь к STL
        stl_path = temp_path.replace(".scad", ".stl")
        return stl_path
    finally:
        # Очистка временных файлов
        os.remove(temp_path)
'''
```

Такой подход позволяет полностью автоматизировать процесс генерации, проверки и экспорта модели без ручного вмешательства.

7. Обработка ошибок и рекурсивное уточнение

Несмотря на тщательный парсинг, генерация кода может иногда приводить к синтаксическим ошибкам, особенно при работе с сложными параметрами. LangChain предоставляет

механизмы для повторных запросов к LLM. Один из вариантов – использовать `Retryable` в цепочке, или вручную отлавливать исключения парсера и отправлять в модель запрос на «исправьте ошибку: {error_message}» с сохранением прежних ограничений.

Пример функции обратного вызова:

```
...  
def handle_error(error_msg: str) -> str:  
    return f"Исправь ошибку: {error_msg}. Сгенерируй корректный код, соблюдая исходные  
параметры."  
  
# Внутри LLMChain можно использовать:  
chain_with_retry = LLMChain(  
    llm=llm,  
    prompt=prompt,  
    output_parser=OpenSCADParser(),  
    verbose=True,  
    callbacks=[handle_error] # упрощённый пример  
)  
...
```

Таким образом, система получает возможность «само-практиковаться», улучшая качество генерируемого кода.

8. Пример практического сценария

Допустим, пользователь хочет создать модуль «шестерню с радиусом 30 мм, 12 зубцами, толщиной 5 мм». Запрос передаётся агенту, который формирует промпт и запрашивает генерацию кода. Ниже полный пример взаимодействия:

```
...  
user_request = "Создай шестиугольный блок с длиной 50 мм, шириной 30 мм и высотой  
20 мм, а также вставь в центр отверстие диаметром 10 мм"  
result = chain.run(user_request)  
parsed_code = OpenSCADParser().parse(result)  
stl_file = execute_openSCAD(parsed_code)  
print(f"Файл STL сохранён в {stl_file}")  
...
```

В результате получаем готовый STL-файл, который можно импортировать в любую программу 3D-визуализации или отправить на 3D-печать.

9. Расширение функциональности

После базовой реализации можно добавить несколько типовых расширений:

- Параметрические модули: Позволить пользователю задавать функции-модули, которые потом могут быть переиспользованы.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.