

Эволюция LangChain

от chains до Deep Agents и Open SWE

Большой tutorial по экосистеме LangChain: chains, LCEL, LangGraph, Deep Agents, Open SWE. Python 1.0+, реальные API, плотный код.

Содержание

- 1** LangChain 1.0 -- chains, LCEL, agents, retrievers
- 2** LangGraph 1.0 -- state, nodes, persistence, HTL
- 3** Deep Agents -- harness, todos, virtual FS, subagents
- 4** Open SWE -- async coding agent, triggers, dashboard
- 5** Экосистема -- LangSmith, Studio, deployment

Всего: 122 content слайдов + cover, TOC, 5 dividers, recap = 132

LangChain 1.0

chains, LCEL, agents, retrievers

Фундамент: композиция через LCEL (`| pipe`), унифицированный `init_chat_model`, output parsers, retrievers, tools, память, middleware. Всё, что нужно, чтобы собрать production-ready LLM-приложение до перехода к LangGraph и Deep Agents.

prompt | model | parser

ChatPromptTemplate

|

ChatModel

|

OutputParser

композиция через LCEL

Что такое LangChain в 2025+

LangChain -- Python/JS фреймворк для сборки LLM-приложений и агентов. С версии 1.0 (релиз 22.10.2025) он построен поверх LangGraph-runtime и позиционируется как "самый быстрый способ собрать агента с любым провайдером моделей".

LCEL

LangChain Expression Language

Декларативная композиция через pipe-оператор. Любая цепочка -- Runnable. invoke/batch/stream/async работают одинаково.

create_agent

унифицированный вход

Один вызов вместо зоопарка legacy agent-типов. Построен на LangGraph -- получаешь durable execution бесплатно.

Middleware

cross-cutting hooks

before_model / after_model / before_tool / after_tool. HITL, PII-редакция, summarization -- first-class встроенные middleware.

~140k звезд GitHub | MIT | Python: langchain 1.3.10 / langchain-core 1.4.8 | JS: @langchain/langchain

pip install -- и сразу в дело

```
// shell/install.sh:1-11
```

1. Core -- обязательно для всех остальных пакетов

```
pip install langchain
```

2. Provider-интеграции -- ставим только те, что нужны

```
pip install langchain-openai
```

```
pip install langchain-anthropic
```

```
pip install langchain-google
```

3. (опционально) дополнительные интеграции

```
pip install langchain-community # ~700 community-пакетов
```

```
pip install langchain-classic # legacy chains/AgentExecutor
```

INFO

langchain-core тянется автоматически как зависимость langchain -- отдельно ставить не нужно.
[sources.python.langchain.com/docs/introduction/#installation](https://python.langchain.com/docs/introduction/#installation)

Первый вызов модели

```
// examples/hello.py:1-11
```

```
# Один универсальный инициализатор для всех провайдеров
from langchain.chat_models import init_chat_model
```

```
# Формат: "<provider>:<model>"
model = init_chat_model("openai:gpt-4.1-mini")
```

```
# invoke -> BaseMessage; нам нужен .content
result = model.invoke("Say hello in one sentence")
print(result.content)
```

```
# >>> "Hello! How can I help you today?"
```

INFO

init_chat_model**del**

Один фабричный
вызов вместо
ChatOpenAI /
ChatAnthropic /

SUCCESS

Что вернется

Объект AIMessage:
content,
response_metadata
(usage,
model_name), id,
tool_calls.

Переключение провайдера -- одна строка

```
// examples/multi_provider.py:1-14

from langchain.chat_models import init_chat_model

# OpenAI
m_openai = init_chat_model("openai:gpt-4.1-mini")

# Anthropic
m_anthropic = init_chat_model("anthropic:claude-3-7-sonnet-latest")

# Google Vertex AI
m_google = init_chat_model("google_vertexai:gemini-2.0-flash")

# Все три -- объекты BaseChatModel. Один и тот же .invoke()
for m in [m_openai, m_anthropic, m_google]:
    print(type(m).__name__, ":", m.invoke("ping").content[:30])
```

WARNING

Нужны API-ключи в env: OPENAI_API_KEY / ANTHROPIC_API_KEY / GOOGLE_API_KEY --

source: python.langchain.com/docs/concepts/chat_models/
провайдер-пакет сам их подхватит.

Стандартизированные сообщения

// examples/messages.py:1-16

```
from langchain.messages import (
    HumanMessage, AIMessage, SystemMessage, ToolMessage,
)
from langchain.chat_models import init_chat_model

model = init_chat_model("openai:gpt-4.1-mini")

messages = [
    SystemMessage(content="You are a concise assistant."),
    HumanMessage(content="What is LCEL?"),
    AIMessage(content="LCEL = pipe-based composition in LangChain."),
    HumanMessage(content="Show a one-line example."),
]

response = model.invoke(messages)
// note: snippet has 16 lines, card fits ~14
print(response.content)
```

INFO

С v1.0 контент -- стандартизированные content blocks: reasoning traces, citations, tool_call

Потоковый вывод -- token за токеном

```
// examples/streaming.py:1-17
```

```
from langchain.chat_models import init_chat_model
```

```
model = init_chat_model("openai:gpt-4.1-mini")
```

```
# .stream() -> итератор по AIMessageChunk
```

```
for chunk in model.stream("Write a haiku about Python"):
```

```
    # У каждого chunk-а есть .content (текст) и .response_metadata
    print(chunk.content, end="", flush=True)
```

```
print() # перевод строки после потока
```

```
# Async-вариант: model.astream() -- то же самое в async-контексте
```

```
import asyncio
```

```
async def main():
```

```
// note: shipper has 17 lines, card fits ~14
```

```
    async for chunk in model.astream("Write a haiku about async"):
```

S U C C E S S

AIMessageChunk можно складывать через оператор + -- для буферизации и метрик.

source: python.langchain.com/docs/how_to/streaming/

ChatPromptTemplate -- декларативно

```
// examples/prompt_basic.py:1-15
```

```
from langchain_core.prompts import ChatPromptTemplate
from langchain.chat_models import init_chat_model

prompt = ChatPromptTemplate.from_messages([
    ("system", "Translate the following text to {language}."),
    ("human", "{text}"),
])

model = init_chat_model("openai:gpt-4.1-mini")

# .invoke() принимает dict и подставляет переменные
messages = prompt.invoke({"language": "French", "text": "Hello world"})
print(messages.to_messages())

# -> [SystemMessage(...), HumanMessage(...)] -- готов к model.invoke()
// note: snippet has 15 lines; card fits ~14
```

INFO

Кортежи ("role", "text") -- шорткат. Для сложного контента передавайте Message-объекты.

source: python.langchain.com/docs/concepts/prompt_templates/

MessagesPlaceholder + few-shot

// examples/prompt_fewshot.py:1-24

```
from langchain_core.prompts import (
    ChatPromptTemplate, MessagesPlaceholder, FewShotChatMessagePromptTemplate,
)

# Few-shot блок: примеры "вопрос -> ответ"
examples = [
    {"input": "2+2", "output": "4"},
    {"input": "3*3", "output": "9"},
]
example_prompt = ChatPromptTemplate.from_messages([
    ("human", "{input}"),
    ("ai", "{output}"),
])
few_shot = FewShotChatMessagePromptTemplate(
    example_prompt=example_prompt, examples=examples,
    // note: snippet has 24 lines, card fits ~14
)
```

S U C C E S S

MessagesPlaceholder("history") -- дырка, в которую при invoke подставляется список
 prompt = ChatPromptTemplate.from_messages([
 source_output_langchain_examples_chat/ how_to_few_shot_examples_chat/
 про систему, которую мы создаем assistant.])

StrOutputParser -- самый частый

```
// examples/parser_str.py:1-17
```

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain.chat_models import init_chat_model
```

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "Translate to French."),
    ("human", "{text}"),
])
model = init_chat_model("openai:gpt-4.1-mini")
```

```
# Склеиваем через LCEL: prompt | model | parser
chain = prompt | model | StrOutputParser()
```

```
# Теперь на выходе str, а не AIMessage
result = chain.invoke({"text": "Hello world"})
// note: snippet has 17 lines, card fits ~14
print(type(result).__name__, "->", result)
```

INFO

StrOutputParser просто достает .content из AIMessage. Без него пришлось бы делать

result.content вручную.

[sources.python.langchain.com/docs/concepts/output_parsers/](https://python.langchain.com/docs/concepts/output_parsers/)

PydanticOutputParser -- типизированный выход

// examples/parser_pydantic.py:1-20

```
from pydantic import BaseModel, Field
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import PydanticOutputParser
from langchain.chat_models import init_chat_model
```

```
class MovieReview(BaseModel):
    title: str = Field(description="Movie title")
    rating: int = Field(description="Rating from 1 to 10")
    summary: str = Field(description="One-sentence summary")
```

```
parser = PydanticOutputParser(pydantic_object=MovieReview)
```

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "Extract review fields.\n{format_instructions}"),
    ("human", "{review_text}"),
    // note: snippet has 20 lines, card fits ~14
]).partial(format_instructions=parser.get_format_instructions())
```

S U C C E S S

В v1.0 рекомендуется model.with_structured_output(schema) -- он использует tool calling и

source: python.langchain.com/docs/how_to/pydantic_output_parser/

```
print(review.title, review.rating, review.summary)
```

with_structured_output -- рекомендованный путь

```
// examples/structured.py:1-17
```

```
from pydantic import BaseModel, Field
from langchain.chat_models import init_chat_model
```

```
class Weather(BaseModel):
    city: str = Field(description="City name")
    temperature_c: float = Field(description="Temperature in Celsius")
    conditions: str = Field(description="Weather summary")
```

```
# Один вызов -- и модель возвращает типизированный Pydantic-объект
model = init_chat_model("openai:gpt-4.1-mini")
structured = model.with_structured_output(Weather)
```

```
result: Weather = structured.invoke("Weather in Paris?")
print(result.city, result.temperature_c, result.conditions)
```

```
// note: snippet has 17 lines, card fits ~14
```

```
# method="json_mode" -- если провайдер не поддерживает tool calling
```

INFO

Под капотом: tool calling или provider-native JSON mode. Без extra LLM-вызовов, в один

проход: [source: python.langchain.com/docs/how_to/structured_output/](https://python.langchain.com/docs/how_to/structured_output/)

| -- декларативная композиция

// examples/lcel_intro.py:1-19

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain.chat_models import init_chat_model
```

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a {style} assistant."),
    ("human", "{question}"),
])
model = init_chat_model("openai:gpt-4.1-mini")
```

```
# prompt, model, parser -- все три реализуют Runnable
# Оператор | склеивает их в один chain
chain = prompt | model | StrOutputParser()
```

```
# type(chain) -> RunnableSequence
// note: snippet has 19 lines, card fits ~14
print(type(chain).__name__)
```

S U C C E S S

Runnable -- единый контракт invoke / batch / stream / ainvoke / abatch / astream / async

source: python.langchain.com/docs/concepts/lcel/

stream:

invoke / batch / stream -- без смены кода

```
// examples/runnable_modes.py:1-16
```

```
chain = prompt | model | StrOutputParser()
```

```
# 1. invoke -- один вход, один выход
```

```
out_one = chain.invoke({"language": "French", "text": "Good morning"})
```

```
# 2. batch -- список входов, список выходов (параллельно)
```

```
out_many = chain.batch([
    {"language": "French", "text": "Good morning"},
    {"language": "German", "text": "Good morning"},
    {"language": "Spanish", "text": "Good morning"},
])
print(out_many) # ["Bonjour", "Guten Morgen", "Buenos dias"]
```

```
# 3. stream -- итератор по чанкам (стримит последний Runnable)
```

```
for chunk in chain.stream({"language": "French", "text": "Stream me"}):
    // note: streamer has 16 lines, card fits ~ 14
    print(chunk, end="", flush=True)
```

INFO

batch полезен для embedding-style задач. stream -- для UX с typewriter-эффектом в UI.

async + config -- полный контроль

// examples/runnable_async.py:1-21

```
import asyncio
from langchain_core.runnables import ConfigurableField

# Любой Runnable имеет ainvoke / astream / abatch
async def main():
    # Один async-вызов
    result = await chain.ainvoke({"language": "French", "text": "Hi"})

    # Async stream
    async for chunk in chain.astream({"language": "French", "text": "Hi"}):
        print(chunk, end="", flush=True)

asyncio.run(main())

# configurable_fields -- параметры, которые можно переопределять
// note: Slippet has 21 lines, card fits ~14
# на лету через config={"configurable": {...}}
```

S U C C E S S

configurable_fields + ConfigurableField -- паттерн для multi-tenant LLM-приложений.

source: python.langchain.com/docs/how_to/configurable/

RunnableLambda + RunnablePassthrough

// examples/runnable_lambda.py:1-19

```
from langchain_core.runnables import RunnableLambda, RunnablePassthrough

# RunnableLambda -- оборачивает произвольную функцию
upper = RunnableLambda(lambda x: x.upper())
word_count = RunnableLambda(lambda text: {"text": text, "words": len(text.split())})

# RunnablePassthrough -- пропускает вход дальше (для branching)
passthrough = RunnablePassthrough()

# Пример: текст -> uppercase -> посчитать слова -> смёрджить с оригиналом
chain = (
    word_count
    | RunnablePassthrough.assign(upper=upper)
)

// note: snippet has 19 lines, card fits ~14
result = chain.invoke("hello world from langchain")
```

INFO

`RunnablePassthrough.assign`, добавляет новые ключи, сохраняя исходные. Идеален для RAG style цепочек.

source: https://python.langchain.com/docs/how_to/functions/

Parallel + Branch -- fan-out и роутинг

```
// examples/runnable_parallel.py:1-17
```

```
from langchain_core.runnables import RunnableParallel, RunnableBranch

# Parallel -- один вход, несколько параллельных Runnable-ов, dict на выходе
joke_chain = ChatPromptTemplate.from_template("Tell a joke about {topic}") | model
poem_chain = ChatPromptTemplate.from_template("Write a poem about {topic}") | model

parallel = RunnableParallel(joke=joke_chain, poem=poem_chain)
result = parallel.invoke({"topic": "cats"})
print(result.keys()) # dict_keys(["joke", "poem"])

# Branch -- if/else роутинг по условию
branch = RunnableBranch(
    (lambda x: "code" in x["topic"].lower(), code_chain),
    (lambda x: "math" in x["topic"].lower(), math_chain),
    general_chain, # default
)
// note: snippet has 17 lines, card fits ~14
```

S U C C E S S

RunnableParallel выполняется параллельно -- отлично для multi-aspect анализа (sentiment + summary + keywords).

source: python.langchain.com/docs/how_to/branching/

Vector store -> retriever -> RAG-цепочка

// examples/retriever_rag.py:1-25

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.runnables import RunnablePassthrough
```

1. Эмбеддинги и индекс

```
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
docs = ["Cats are mammals.", "Python is a programming language.",
        "LangChain helps build LLM apps."]
vectorstore = FAISS.from_texts(docs, embedding=embeddings)
```

2. .as_retriever() превращает store в Runnable

```
retriever = vectorstore.as_retriever(search_kwargs={"k": 2})
```

3. RAG-цепочка через LCEL: context + question -> answer

```
from langchain_core.prompts import ChatPromptTemplate
// note: snippet has 25 lines, card fits ~14
from langchain.chat_models import init_chat_model
```

INFO

Альтернативные store: Chroma (легковесный), PGVector (production Postgres), Pinecone, Weaviate, Qdrant.

source: python.langchain.com/docs/concepts/retrievers/

@tool -- любая функция становится ИНСТРУМЕНТОМ

// examples/tool_basic.py:1-20

```
from langchain.tools import tool
from langchain.chat_models import init_chat_model

@tool
def get_weather(city: str) -> str:
    """Get the current weather for a given city."""
    return f"Sunny, 22C in {city}"

@tool
def search_docs(query: str, top_k: int = 3) -> list[str]:
    """Search internal documentation. Returns top_k snippets."""
    return [f"doc about {query} #{i}" for i in range(top_k)]

# bind_tools -- модель знает, какие функции можно вызывать
model = init_chat_model("openai:gpt-4.1-mini")
// note: snippet has 20 lines, card fits ~14
bound = model.bind_tools([get_weather, search_docs])
```

S U C C E S S

docstring инструмента = описание, которое видит модель. Названия параметров должны совпадать с параметрами функции.

```
source: https://langchain.com/docs/how_to/tool_calling/
On [[ name="get_weather", args: { city: "Paris", "id": "..."}]]
```

create_agent -- один вход для всех агентов

// examples/agent_basic.py:1-19

```
from langchain.agents import create_agent
from langchain.tools import tool
```

```
@tool
def get_weather(city: str) -> str:
    """Get weather for a city."""
    return f"Sunny, 22C in {city}"
```

Один вызов вместо create_react_agent / create_openai_functions_agent / ...

```
agent = create_agent(
    model="openai:gpt-4.1",
    tools=[get_weather],
    system_prompt="You are a weather assistant.",
)
```

// note: snippet has 19 lines, card fits ~14

invoke -> dict {'messages': [...]}; последнее сообщение = ответ

INFO

Под капотом LangGraph runtime: durable execution, checkpointing, human-in-the-loop доступны через middleware.

source: <https://docs.langchain.com/oss/python/langchain/agents>

Краткосрочная память: thread + checkpointer

```
// examples/memory_short.py:1-21

from langchain.agents import create_agent
from langgraph.checkpoint.memory import InMemorySaver

checkpointer = InMemorySaver()

agent = create_agent(
    model="openai:gpt-4.1-mini",
    tools=[...],
    checkpointer=checkpointer, # <-- persistent state
)

config = {"configurable": {"thread_id": "user-42"}}

# Первый turn
agent.invoke({"messages": [{"role": "user",
// note: shipper has 21 lines, card fits ~14
"content": "My name is Alice."}]}], config=config)
```

S U C C E S S

InMemorySaver для dev. Для прода -- PostgresSaver / RedisSaver (те же LangGraph checkpointer).
 source: https://github.com/langchain-ai/langgraph/blob/main/langgraph/persistence/in_memory.py

Долгосрочная память: store + namespace

// examples/memory_long.py:1-21

```
from langgraph.store.memory import InMemoryStore
from langchain.agents import create_agent
from langchain.embeddings import init_embeddings

# Store может быть in-memory (dev) или Postgres (prod)
store = InMemoryStore(
    index={"embed": init_embeddings("openai:text-embedding-3-small"),
          "dims": 1536},
)

agent = create_agent(
    model="openai:gpt-4.1-mini",
    tools=[...],
    store=store,
```

//note: snippet has 21 lines, card fits ~14

INFO

Long-term memory переживает thread. Идеальна для user preferences, summary прошлых сессий, learned facts.

Cross-cutting concerns одной строкой

// examples/middleware.py:1-22

```
from langchain.agents import create_agent
from langchain.agents.middleware import (
    HumanInTheLoopMiddleware,
    PIIRedactionMiddleware,
    SummarizationMiddleware,
)
from langchain.tools import tool
```

@tool

```
def send_email(to: str, body: str) -> str:
    """Send an email to recipient."""
    return f"sent to {to}"
```

```
agent = create_agent(
    model="openai:gpt-4-1"
```

// note. snippet has 22 lines, card fits ~14
tools=[send_email],

WARNING

Все три middleware – встроенные. Custom middleware пишутся как before_model/after_model

source: <https://langchain.com/docs/python/langchain/middleware>
Hooks SummarizationMiddleware(trigger=(tokens, 4000)),

Что нового в 1.0 vs 0.3

0.x (legacy)

- create_react_agent / create_openai_functions_agent / create_structured_chat_agent
- LLMChain, RetrievalQA, ConversationalRetrievalQA
- ChatOpenAI / ChatAnthropic / ChatGoogleGenerativeAI отдельно
- Кастомные callbacks для HITL и PII
- Verbose LLMChain с ручным manage_prompts

1.0 (current)

- + create_agent -- единая точка входа (построен на LangGraph)
- + Middleware-система: HITL, PII, Summarization как first-class
- + init_chat_model("<provider>:<model>") -- один инициализатор
- + with_structured_output -- tool calling / provider-native JSON
- + Семантическое версионирование: до 2.0 breaking changes не будет
- + langchain-classic -- пакет для обратной совместимости legacy chains

INFO

Миграция: `pip install langchain-classic` legacy LLMChain/AgentExecutor работают, но новый

JS-аналог: что есть, чего нет

// examples/ts_basic.ts:1-16

```
import { initChatModel } from "langchain/chat_models/universal";
import { createAgent } from "langchain/agents";
import { tool } from "@langchain/core/tools";
import { z } from "zod";

const getWeather = tool(
  async ({ city }) => `Sunny, 22C in ${city}`,
  { name: "get_weather", description: "Get weather",
    schema: z.object({ city: z.string() }) },
);

const model = await initChatModel("openai:gpt-4.1");
const agent = createAgent({ model, tools: [getWeather] });
const result = await agent.invoke({
  messages: [{ role: "user", content: "weather in Paris?" }],
});
```

// note. snippet has 16 lines, card fits ~14

Что есть в @langchain/langchain:

- + initChatModel
- + createAgent
- + @langchain/core (tools, prompts, parsers)
- + LCEL и Runnable-протокол
- + Стандартизированные Messages
- + Vector store интеграции

Чего нет или слабее:

- langchain-classic покрытие уже
- Часть community-интеграций
- Tracing middleware меньше

Когда LangChain 1.0 -- правильный выбор

Production-ready commitment, единая точка входа для агентов, встроенные middleware. Но абстракция скрывает LangGraph -- если нужен fine-grained контроль над execution, придется проваливаться глубже.

PROS

- + Семантическая стабильность -- обязательство не ломать API до 2.0
- + create_agent вместо зоопарка agent-типов
- + init_chat_model -- переключение провайдера без рефакторинга
- + Middleware-система (HITL, PII, Summarization) из коробки
- + LangGraph-runtime под капотом -- durable execution бесплатно
- + ~700+ интеграций через community-

CONS

- Кривая обучения для middleware (before/after hooks)
- Часть экосистемы переехала в langchain-classic -- миграционная боль
- Абстракция скрывает LangGraph -- для fine-grained нужен fallback
- Bundled-версии зависимостей (langchain-openai, -anthropic, ...) нужно фиксировать
- Исторически bloated core -- в 1.0 стало lean, но восприятие осталось

Дальше: LangGraph -- явный control flow

LangChain 1.0 -- это "правильный путь по умолчанию" для большинства задач. Когда LCEL-pipeline перестает хватать (ветвления, циклы, человеческое одобрение, persistence) -- под капотом работает LangGraph. В следующей секции разберем его как самостоятельный runtime: граф, узлы, edges, checkpoint, human-in-the-loop как first-class концепции.

Граф вместо пайплайна

StateGraph, узлы (nodes), ребра (edges), условные переходы. Полный контроль над execution.

Persistence

Checkpointer, thread_id, time-travel. State между turn-ами хранится на диске.

Human-in-the-loop

interrupt_before / interrupt_after узлов. Апрув через Command. Не нужен middleware-хак.

2

SECTION 2: LANGGRAPH 1.0

LangGraph 1.0: stateful runtime

State, nodes, persistence, HITL. Production-ready durable execution: агенты, которые переживают падение сервера и одобряются человеком.

Зачем LangGraph: что не может обычный LangChain

INFO

LCEL хорош для

- простых pipeline (prompt | model | parser)
- одного прохода без ветвлений
- read-only чатов без state между вызовами

WARNING

LCEL не даёт

- циклов и произвольной топологии графа
- first-class persistence и time-travel
- настоящей паузы на human approval
- multi-agent и параллельных веток (Send)
- восстановления после падения посреди long-run

Установка: одна команда

```
pip install -U langgraph
```

```
# extras: postgres / sqlite checkpointers -- отдельные пакеты
```

```
pip install -U langgraph langgraph-checkpoint-postgres
```

```
pip install -U langgraph langgraph-checkpoint-sqlite
```

```
# JS / TypeScript
```

```
npm install @langchain/langgraph @langchain/langgraph-checkpoint
```

SUCCESS

Что в коробке

state, persistence, HITL, streaming, ToolNode, subgraph API. Никаких внешних сервисов кроме самого рантайма.

State как TypedDict -- первая итерация

```
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

> class State(TypedDict):
>     question: str
>     answer: str
>     steps: int

def answer(state: State) -> dict:
    return {"answer": f"echo: {state['question']}", "steps": 1}

builder = StateGraph(State)
builder.add_node("answer", answer)
builder.add_edge(START, "answer")
builder.add_edge("answer", END)
graph = builder.compile()

>
> print(graph.invoke({"question": "hi", "steps": 0}))
// note: snippet has 19 lines, card fits ~17
> # -> {'question': 'hi', 'answer': 'echo: hi', 'steps': 1}
```

Annotated + add_messages -- каналы с reducer

```
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

class State(TypedDict):
    # reducer add_messages склеивает списки сообщений по правилам
    > messages: Annotated[list, add_messages]

def echo(state: State):
    last = state["messages"][-1]
    > return {"messages": [{"role": "assistant", "content": f"echo: {last.content}"}]}
    >
    g = StateGraph(State)
    g.add_node("echo", echo)
    g.add_edge(START, "echo")
    g.add_edge("echo", END)
    app = g.compile()

// note: snippet has 20 lines, card fits ~18
print(app.invoke({"messages": [{"role": "user", "content": "hi"}]}))
```

operator.add -- аккумуляция для list-канала

```
import operator
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

class State(TypedDict):
    # каждый узел возвращает кусок списка, operator.add склеивает
    log: Annotated[list[str], operator.add]
    tokens: Annotated[int, operator.add]
    >
    >

def step(state: State):
    return {"log": ["step-1"], "tokens": 12}

def another(state: State):
    return {"log": ["step-2"], "tokens": 7}

g = StateGraph(State)
g.add_node("a", step)
g.add_node("b", another)
g.add_edge(START, "a")
g.add_edge("a", "b")
g.add_edge("b", END)
```

// code snippet has 25 lines, card fits ~18

dataclass + LastValue -- когда хочется типизации

```

from dataclasses import dataclass, field
from typing import Annotated
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

@dataclass
class State:
    question: str = ""
> # dataclass + Annotated -- каналы работают точно так же
> messages: Annotated[list, add_messages] = field(default_factory=list)
    approved: bool = False

def greet(state: State):
    return {"messages": [{"role": "assistant", "content": f"hi, {state.question}"]}]

g = StateGraph(State)
g.add_node("greet", greet)
g.add_edge(START, "greet")
g.add_edge("greet", END)
// code snippet has 24 lines, but fits ~18
app = g.compile()
print(app.invoke(State(question="alex")))

```

StateGraph: builder.compile() -- базовый граф

```
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

class State(TypedDict):
    n: int

def inc(state: State):
    return {"n": state["n"] + 1}

builder = StateGraph(State)
builder.add_node("inc", inc) # регистрируем узел
> builder.add_edge(START, "inc") # поток входа
> builder.add_edge("inc", END) # поток выхода
>
graph = builder.compile() # компиляция -- граф готов к invoke

> print(graph.invoke({"n": 0})) # -> {'n': 1}
```

START, END и поток данных через рёбра

```
# START и END -- это специальные sentinel-узлы, не callable
from langgraph.graph import StateGraph, START, END
```

```
>
```

```
class State(TypedDict):
    text: str
```

```
def upper(state: State):
    return {"text": state["text"].upper()}
```

```
def exclaim(state: State):
    return {"text": state["text"] + "!"}
```

```
g = StateGraph(State)
> g.add_node("upper", upper)
g.add_node("exclaim", exclaim)
> g.add_edge(START, "upper")      # вход в граф
> g.add_edge("upper", "exclaim")  # внутреннее ребро
g.add_edge("exclaim", END)        # выход из графа
```

```
// note: snippet has 21 lines, card fits ~18
```

```
app = g.compile()
print(app.invoke({"text": "hi"})) # -> {"text": "HI!"}
```

source: langchain-ai.github.io/langgraph/concepts/low_level/#why-langgraph

Command -- узел сам решает, куда идти дальше

```
from langgraph.graph import StateGraph, START, END
from langgraph.types import Command
from typing_extensions import TypedDict
from typing import Literal
class State(TypedDict):
    n: int
> def decide(state: State) -> Command[Literal["inc", "halt"]]:
>     # Command(update, goto) -- обновляет state и сам выбирает узел
    if state["n"] < 3:
>         return Command(update={"n": state["n"] + 1}, goto="inc")
    return Command(update={}, goto="halt")
g = StateGraph(State)
g.add_node("decide", decide)
g.add_node("inc", inc)
g.add_node("halt", lambda s: s)
g.add_edge(START, "decide")
g.add_edge("inc", "decide")
g.add_edge("halt", END)
app = g.compile()
// app: 50 lines, card fits ~18
print(app.invoke({"n": 0}))
```


Conditional edges: routing по содержимому

```
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
```

```
class State(TypedDict):
    needs_tool: bool
    answer: str
```

```
def agent(state: State) -> dict:
    return {"answer": "model-output"}
```

```
def tool_node(state: State) -> dict:
    return {"answer": "tool-output"}
```

```
def route(state: State) -> str:
    # возвращаем ключ, который есть в path_map ниже
    return "tool_node" if state["needs_tool"] else END
```

```
g = StateGraph(State)
```

```
> g.add_node("agent", agent)
# Note: Shipped as 3 lines, can be 1-18
```

```
> g.add_node("tool_node", tool_node)
```

```
> g.add_edge(START, "agent")
```

```
> g.add_conditional_edges("agent", route, {
```

Циклы: agentic loop без рекурсии в коде

```
# агентный цикл: agent <-> tools, выход когда done=true
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
```

```
class State(TypedDict):
    done: bool
    iter: int
```

```
def agent(state: State) -> dict:
    return {"iter": state["iter"] + 1, "done": state["iter"] >= 3}
```

```
def maybe_continue(state: State) -> str:
    return "agent" if not state["done"] else END
```

```
> g = StateGraph(State)
> g.add_node("agent", agent)
g.add_edge(START, "agent")
g.add_conditional_edges("agent", maybe_continue, {"agent": "agent", END: END})
```

```
// app: suppress 2 lines, card fits ~18
```

```
app = g.compile()
print(app.invoke({"done": False, "iter": 0}))
```

```
# agent крутится 3 раза, потом уходит в END
```

source: [langchain-ai.github.io/langgraph/concepts/low_level/#cycles](https://github.com/langchain-ai/langgraph/concepts/low_level/#cycles)

InMemorySaver + thread_id: stateful сессии

```

from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langgraph.checkpoint.memory import InMemorySaver

class State(TypedDict):
    messages: Annotated[list, add_messages]

def echo(state: State):
    last = state["messages"][-1]
    return {"messages": [{"role": "assistant", "content": f"echo: {last.content}"]}

g = StateGraph(State)
g.add_node("echo", echo)
g.add_edge(START, "echo")
g.add_edge("echo", END)

# checkpointer обязательно для thread persistence
// note: snippet has 26 lines, code has 13
checkpointer = InMemorySaver()
app = g.compile(checkpointer=checkpointer)

```

SqliteSaver и PostgresSaver для production

```
> # SQLite -- файл на диске, идеален для dev / small-  
prod  
from langgraph.checkpoint.sqlite import SqliteSaver  
>  
> with SqliteSaver.from_conn_string("./checkpoints.db")  
as cp:  
> app = g.compile(checkpointer=cp)  
cfg = {"configurable": {"thread_id": "u1"}}  
app.invoke({"messages": []}, cfg)
```

данные переживают рестарт процесса.
Для asyncio-варианта -- aiosqlite.

```
> # Postgres -- production-grade, multi-instance  
from langgraph.checkpoint.postgres import  
PostgresSaver  
>  
DB = "postgresql://user:pass@host:5432/lg"  
with PostgresSaver.from_conn_string(DB) as cp:  
> # первый запуск создаст schema  
cp.setup()  
app = g.compile(checkpointer=cp)  
cfg = {"configurable": {"thread_id": "u1"}}  
app.invoke({"messages": []}, cfg)
```

StateSnapshot -- что лежит в checkpoint

```
# После invoke можно достать текущий snapshot:  
> snapshot = app.get_state(cfg)  
  
# snapshot -- это StateSnapshot с полями:  
> # .config -- thread_id + checkpoint_id  
> # .metadata -- step, source, writes  
> # .values -- текущие значения всех каналов state  
> # .next -- tuple узлов, которые будут выполняться следующими  
> # .tasks -- PregelTask с pending/result/error  
  
print(snapshot.next) # () -- граф завершён  
print(snapshot.values["messages"][-1].content)
```

get_state_history: вся история шагов

```
# Каждый super-step -- отдельный checkpoint в thread
> history = list(app.get_state_history(cfg))
>
> # history[0] -- последний шаг (самый свежий)
# history[-1] -- самый первый (начало сессии)

> for i, snap in enumerate(history):
    print(i, snap.metadata.get("step"), snap.values.get("iter"))

> # replays = форк от любого прошлого snapshot:
> old = history[2].config
app.invoke(None, old) # переигрывает только следующие шаги
```

update_state: форкнуть состояние и пойти другой веткой

```
# patch значения канала прямо в snapshot -- создаётся новый checkpoint
> app.update_state(
>   cfg,
>   values={"messages": [{"role": "user", "content": "rewind"}]},
>   as_node="user_input", # от имени какого узла пишем
> )

# Дальше invoke(None, cfg) переигрывает граф с нового состояния
app.invoke(None, cfg)
>
> # Типичный приём: "что если пользователь сказал не X, а Y?" --
# ответвляемся, смотрим альтернативный прогон без потери истории.
```

interrupt + Command(resume=): пауза на человеке

```

from langgraph.types import interrupt, Command
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import InMemorySaver
from typing_extensions import TypedDict
class State(TypedDict):
    question: str
    approved: bool
def ask(state: State):
    # interrupt() -- пауза. Возвращает значение из Command(resume=...)
> answer = interrupt({"question": "Approve sending this message?"})
    return {"approved": answer == "yes"}
g = StateGraph(State)
g.add_node("ask", ask)
g.add_edge(START, "ask")
g.add_edge("ask", END)
app = g.compile(checkpointer=InMemorySaver())
> cfg = {"configurable": {"thread_id": "approval-1"}}
> app.invoke({"question": "send email", "approved": False}, cfg) # пауза
> result = app.invoke(Command(resume="yes"), cfg) # resume
# result["approved"] == True
print(result["approved"]) # -> True

```


Как выглядит HITL-цикл снаружи

INFO

Серверная сторона

1. `graph.invoke(input, cfg)`
2. Внутри узла вызван `interrupt(payload)`
3. Граф замораживается, состояние в `checkpoint`
4. Возвращается `GraphInterrupt` с `payload`
5. Сервер ждёт -- пользователь думает часы/дни

SUCCESS

Клиентская сторона

1. UI получает `payload`, рисует форму
2. Пользователь жмёт `Approve / Reject`
3. UI делает `POST /threads/{id}/resume`
4. Сервер вызывает `graph.invoke(Command(resume=answer), cfg)`
5. Граф оживает с того же узла

Multi-turn approval: узел review

```
# Узел review -- маршрутизация по ответу человека
from langgraph.types import interrupt, Command
from typing_extensions import TypedDict
class State(TypedDict):
    plan: str
    executed: bool
def plan(state: State):
    return {"plan": "step-A; step-B; step-C"}
> def review(state: State) -> Command:
>     # interrupt() возвращает ответ из Command(resume=...)
>     decision = interrupt({"plan": state["plan"]})
>     if decision == "approve":
>         return Command(goto="execute")
>     if decision == "interrupt":
>         return Command(resume="review")
```

INFO

Полный пример со сборкой графа -- на следующем слайде.

```
return {"executed": True}
```

Multi-turn approval: сборка графа

```
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import InMemorySaver
# plan, review, execute -- из предыдущего слайда
g = StateGraph(State)
> g.add_node("plan", plan)
> g.add_node("review", review)
  g.add_node("execute", execute)
> g.add_edge(START, "plan")
  g.add_edge("plan", "review")
  g.add_edge("execute", END)
app = g.compile(checkpointer=InMemorySaver())

# Цикл: каждый review -- это пауза; Command(goto=...) -- ответвление
> # plan -> review -> (approve: execute | abort: END | else: plan)
> cfg = {"configurable": {"thread_id": "u1"}}
  app.invoke({"plan": "", "executed": False}, cfg)      # пауза 1
  app.invoke(Command(resume="approve"), cfg)            # resume -> execute
```

Subgraphs: композиция и изоляция state

```
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
```

```
class SubState(TypedDict):
    internal: str
```

```
def inner(state: SubState):
    return {"internal": "sub-output"}
```

Собираем subgraph -- у него свой state

```
> sub = StateGraph(SubState)
> sub.add_node("inner", inner)
> sub.add_edge(START, "inner")
> sub.add_edge("inner", END)
sub_compiled = sub.compile()
```

Вставляем как обычный узел в родительский граф

```
class ParentState(TypedDict):
    out: str
// note: snippet has 26 lines, card fits ~18
```

```
> parent = StateGraph(ParentState)
> parent.add_node(sub_block, sub_compiled) # <- subgraph целиком
```

Пять режимов stream_mode

```
cfg = {"configurable": {"thread_id": "u1"}}
# values -- весь state после каждого super-step
> for snap in app.stream({"messages": []}, cfg, stream_mode="values"):
    print(snap["messages"][-1].content)
# updates -- delta: только что вернул каждый узел
> for upd in app.stream({"messages": []}, cfg, stream_mode="updates"):
    print(upd)
# events -- низкоуровневые события (start, end, error, interrupt)
> for ev in app.stream({"messages": []}, cfg, stream_mode="events"):
    print(ev["event"], ev["name"])
# messages -- токены LLM по мере генерации
> for tok, meta in app.stream({"messages": []}, cfg, stream_mode="messages"):
    print(tok.content, end="|")
# custom -- только то, что узлы пишут через get_stream_writer()
> for chunk in app.stream({"messages": []}, cfg, stream_mode="custom"):
    print(chunk)
```

Custom stream: пишем из узла как хотим

```
from langgraph.graph import StateGraph, START, END
from langgraph.config import get_stream_writer
from typing_extensions import TypedDict

class State(TypedDict):
    total: int

def progress(state: State):
    writer = get_stream_writer()    # доступен только во время выполнения узла
    > for i in range(3):
    >     writer({"progress": i, "phase": "thinking"})
    > return {"total": 3}

g = StateGraph(State)
g.add_node("progress", progress)
g.add_edge(START, "progress")
g.add_edge("progress", END)
app = g.compile()

➤ note: snippet has 22 lines, card fits ~18
> # в UI -- только custom-чанки, без промежуточного state
for chunk in app.stream({"total": 0}, stream_mode="custom"):
    print(chunk)
```

Tool calling: tools + LLM binding

```
from langchain_openai import ChatOpenAI
from langchain.tools import tool
from langgraph.graph.message import add_messages
from typing import Annotated
from typing_extensions import TypedDict
@tool
def add(a: int, b: int) -> int:
    "Add two numbers."
    return a + b
tools = [add]
> llm = ChatOpenAI(model="gpt-4o-mini").bind_tools(tools)
class State(TypedDict):
    messages: Annotated[list, add_messages]
    # ... (other state variables)
```

INFO

Сборка графа и запуск -- на следующем слайде.

```
return "tools" if getattr(last, "tool_calls", None) else END
```

Tool calling: сборка графа и запуск

```
from langgraph.graph import StateGraph, START, END
from langgraph.prebuilt import ToolNode
# tools, llm, agent, route -- из предыдущего слайда
g = StateGraph(State)
g.add_node("agent", agent)
> g.add_node("tools", ToolNode(tools))
g.add_edge(START, "agent")
g.add_conditional_edges("agent", route, {"tools": "tools", END: END})
g.add_edge("tools", "agent")
> app = g.compile()

# Цикл: agent решает -> tools исполняет -> agent снова читает результат
> result = app.invoke({
>   "messages": [{"role": "user", "content": "What is 2 + 3?"}]
> })
print(result["messages"][-1].content)
```


LangGraph Studio: визуальный debugger

INFO

Что это

Desktop IDE и web-UI для отладки графов. Показывает граф, каждый super-step, state на каждом шаге, время на узел.

SUCCESS

Что умеет

- запускать граф интерактивно
- ставить breakpoints на узлах
- модифицировать state вручную
- редактировать узлы и перезапускать
- экспорт trace в LangSmith

запуск: `langgraph dev` -- поднимает Studio на `http://localhost:8123`

Deploy через LangGraph Platform

```
# langgraph.json -- declarative config
> {
  "graphs": {"agent": "./agent.py:graph"},
  "env": "./.env",
  "python_version": "3.11"
}
```

```
# CLI: локальный dev-сервер и deploy
langgraph dev    # локальный API + Studio
> langgraph up   # Docker-compose stack (Redis + API + Studio)
langgraph deploy # пуш в LangGraph Platform (managed)
```

Что нового в LangGraph 1.0: четыре фичи

SUCCESS

Durable execution

Состояние персистируется автоматически.
Падение сервера посреди long-run -- граф
восстанавливается ровно с точки
остановки.

INFO

Built-in persistence

InMemorySaver / SqliteSaver /
PostgresSaver -- first-class контракты, а не
отдельный набор фич.

SUCCESS

HITL first-class

interrupt() -- стабильный API, multi-day
approval workflows без костылей.

WARNING

Breaking changes

- langgraph.prebuilt.create_react_agent ->
langchain.agents.create_agent
- MemorySaver -> InMemorySaver (новый
alias)
- semver: стабильно до 2.0

@langchain/langgraph -- JS/TS-аналог

```
> import { StateGraph, START, END, Annotation } from "@langchain/langgraph";
> import { MemorySaver } from "@langchain/langgraph-checkpoint";
```

```
const State = Annotation.Root({
  messages: Annotation({
    reducer: (a, b) => a.concat(b),
    default: () => [],
  }),
});
```

```
const g = new StateGraph(State)
  .addNode("echo", (s) => ({
    messages: [{ role: "assistant", content: "echo: " + s.messages.at(-1).content }],
  }))
  .addEdge(START, "echo")
  .addEdge("echo", END);
```

```
const app = g.compile({ checkpointer: new MemorySaver() });
> const cfg = { configurable: { thread_id: "t1" } };
// note: snippet has 20 lines, here is the first 10
const result = await app.invoke({ messages: [{ role: "user", content: "hi" } ] }, cfg);
```

Когда LangGraph, когда остаться на LCEL

PROS

- + Агент крутится в цикле (tool calls + retry)
- + Нужна пауза на human approval / review
- + Long-running workflow переживает рестарт
- + Сложная топология: ветвления, merge, map-reduce
- + Multi-agent: subgraphs + Send
- + Time-travel и replay для отладки

CONS

- Простой pipeline prompt | model | parser
- Один проход без state между вызовами
- Read-only чат без persistence
- Быстрый прототип без долгоживущего state
- Команда не готова к concepts: channels/reducers/Send

Мостик к Deep Agents

INFO

Что мы только что разобрали

- State, nodes, edges, persistence
- HITL через interrupt + Command(resume)
- Subgraphs, streaming, ToolNode
- Deploy через LangGraph Platform

SUCCESS

Что дальше (Section 3)

Deep Agents -- это готовая архитектура поверх LangGraph: planning tool, filesystem, subagents, middleware.
create_deep_agent -- одна функция вместо сотни строк boilerplate.

WARNING

Связь

create_deep_agent из deepagents==0.6.11 -- это обёртка, которая собирает граф LangGraph с planning tool, файловой системой и subagents. Внутри всё, что мы видели: StateGraph, Command, interrupt, subgraphs.

0

3

SECTION 3

Deep Agents 1.0

Batteries-included agent harness: planning tool, virtual filesystem,
subagents with isolated context, pluggable backends,
HITL middleware.

Built on LangGraph + LangChain 1.0 middleware.
Inspired by Claude Code,
Deep Research, Manus.

LangGraph limits: why a new layer

WARNING

LangGraph is a runtime, not an agent harness.

You still have to wire planning, filesystem access, subagent isolation, HITL approval, and context offload yourself.

PROS

- + Full control over graph topology, cycles, parallel branches (Send)
- + Durable execution, checkpointing, interrupt-based HITL are first-class
- + Stable public API until 2.0 (released 22 Oct 2025)

CONS

- Boilerplate-heavy: planning tool, fs tools, subagent wiring -- all manual
- No built-in context overflow strategy (every tool result floods the main thread)
- No opinionated "coding/research" agent defaults -- you re-implement Claude Code patterns

Deep Agents: opinionated defaults

INFO

Analogy: Django over raw WSGI

Same runtime (LangGraph), but with conventions and pre-built middleware.

// examples/hello.py:1-14

```
# One import, one call -- you get:
# - write_todos planning tool
# - ls / read_file / write_file / edit_file
# - glob, grep, execute (bash)
# - task tool for subagents
# - summarization middleware
```

```
from deepagents import create_deep_agent
```

```
>
> agent = create_deep_agent(
// note: snippet has 14 lines, card fits ~10
>     model="openai:gpt-4.1",
>     tools=[my_tool],
>     system_prompt="...",
```

source: docs.langchain.com/oss/python/deepagents/overview

SUCCESS

Stack

LangGraph (runtime)
 -> create_agent (LangChain 1.0, thin harness)
 -> create_deep_agent (opinionated harness)

Built-in: planning, FS, subagents, HITL, skills.

pip install deepagents

// setup.sh:1-5

```
pip install deepagents
```

```
# or, with uv:
```

```
uv add deepagents
```

// note: snippet has 5 lines, card fits ~3

```
# JS analogue:
```

```
npm install deepagents
```

INFO

Latest stable (snapshot 2026-06-22)

Python: deepagents 0.6.11 (no formal 1.0 yet).

Repo: github.com/langchain-ai/deepagents (~24.9k stars)

WARNING

Dependencies

deepagents pulls in langchain, langgraph, langchain-core.

API keys via env vars: OPENAI_API_KEY, ANTHROPIC_API_KEY, etc.

create_deep_agent: hello world

// examples/hello.py:1-12

```
from deepagents import create_deep_agent

> agent = create_deep_agent(
>   model="openai:gpt-4.1",
>   tools=[],
>   system_prompt="You are a helpful assistant.",
> )

result = agent.invoke({
    "messages": "Write a haiku about Python"
})
print(result["messages"][-1].content)
```

INFO

What you get for free

- write_todos tool
- ls / read_file / write_file / edit_file
- glob, grep, execute (bash)
- task tool for subagents
- summarization middleware
- same LangGraph runtime as create_agent

create_deep_agent: full API

// examples/full_api.py:1-15

```
from deepagents import create_deep_agent
from deepagents.backends import FilesystemBackend

> agent = create_deep_agent(
>   model="anthropic:claude-sonnet-4-5",
>   tools=[my_tool],
>   system_prompt="...",
>   subagents=[researcher, writer],
>   skills=["./skills/review.md"],
>   backend=FilesystemBackend("./ws"),
>   middleware=[my_hitl, my_logger],
>   checkpointer=InMemorySaver(),
>   store=InMemoryStore(),
>   interrupt_on={"bash": True},
> )
```

System prompt: how to talk to a deep agent

```
// examples/system_prompt.py:1-16
```

```
SYSTEM_PROMPT = """
```

```
You are a senior backend engineer.
```

```
> WORKFLOW:
```

- > 1. Plan with write_todos before non-trivial work.
- > 2. Explore the codebase via ls / glob / grep first.
- > 3. Use edit_file for surgical changes, write_file for new files.
- > 4. Delegate research tasks to the "researcher" subagent.
- 5. Run tests via execute; never claim success without output.

```
CONSTRAINTS:
```

- > - Do not modify files outside ./src.
- > - Stop and ask the user if requirements are ambiguous.

```
"""
```

```
agent = create_deep_agent(model=..., system_prompt=SYSTEM_PROMPT)
```

Built-in fs + planning tools

INFO

Eight opinionated defaults, all active unless you override.

Planning, filesystem, search, shell, and subagent delegation -- one import, zero setup.

write_todos

plan / decompose a task into ordered steps

ls

list directory entries in the virtual FS

read_file

read one or many files with line offsets

write_file

create or overwrite a file in the FS

edit_file

surgical string-replace edits (matches all occurrences)

glob

find files by pattern (e.g. `**/*.py`)

grep

regex search across files with context

execute

run a shell command (sandboxed if a backend enforces it)

write_file and edit_file in action

```
// examples/fs_tools.py:1-11
```

```
# The agent calls these tools by name;  
# you describe the goal in the prompt.
```

```
> PROMPT = """  
> 1. Write src/hello.py with a greet(name) function.  
> 2. Use edit_file to add a docstring to greet().  
> 3. Use write_file to add tests/test_hello.py.  
"""
```

```
agent = create_deep_agent(model="openai:gpt-4.1")  
agent.invoke({"messages": PROMPT})
```

SUCCESS

Context overflow protection

Tool outputs larger than a threshold are offloaded to the virtual FS

and replaced with a path + summary in the message history. The agent can re-read specific portions on demand. This is what lets deep agents handle large repos without blowing the context window.

write_todos: explicit planning inside the graph

INFO

write_todos is just a tool, like any other.

The LLM emits a structured plan, the graph stores it in state["todos"],

```
// examples/write_todos.py:1-12
```

```
write_todos(  
> todos=[  
>     {"content": "Read repo structure", "status": "in_progress",  
>      "activeForm": "Reading repo structure"},  
>     {"content": "Implement greet()", "status": "pending",  
>      "activeForm": "Implementing greet()"},  
>     {"content": "Add pytest cases", "status": "pending",  
>      "activeForm": "Adding pytest cases"},  
> ]  
)
```

// note: snippet has 12 lines, card fits ~10

```
# Status: pending | in_progress | completed  
# activeForm: present-continuous shown in UI
```


Plan, Act, Reflect loop

1. PLAN

write_todos:
- decompose task
- order steps
- mark in_progress



2. ACT

execute tool call
(read_file, edit_file, ...)
or delegate via task



3. REFLECT

update todo status
re-plan if blocked
log progress

loop until all todos = completed

INFO

The graph state carries the plan -- every LLM call sees it in the system message, so the agent self-monitors progress across turns.

task tool: delegate, isolate, return

INFO

Why subagents?

Long tool outputs and exploratory searches pollute the main thread.

```
// examples/task_call.py:1-10
```

```
# When the main agent emits a tool call like:
> task(
>   subagent_type="researcher",
>   description="Find papers on RAG evaluation",
>   prompt="Search arXiv for 2025-2026 RAG evaluation surveys.
>   Return a 150-word summary with 3 citations.",
> )

# Deep Agents spins up a fresh deep agent with the researcher profile,
# runs it to completion, and returns only the final message.
```

Subagents: minimal example

// examples/subagents.py:1-23

```
from deepagents import create_deep_agent
```

```
researcher = {
```

```
> "name": "researcher",
```

```
> "description": "Does deep web research, returns citations",
```

```
> "system_prompt": "You are a research specialist. Always cite sources.",
```

```
> "tools": [web_search], # optional, can be []
```

```
}
```

```
writer = {
```

```
> "name": "writer",
```

```
> "description": "Polishes prose into a final report",
```

```
> "system_prompt": "You are a writing specialist.",
```

```
> "tools": [],
```

```
}
```

// note: snippet has 23 lines, card fits ~16

```
agent = create_deep_agent(
```

```
    model="openai:gpt-4.1",
```

```
> tools=[],
```

```
source> subagents=[researcher, writer], ADME
```

```
< )
```

Isolated context for subagents

MAIN AGENT

```
context = [  
  user task,  
  summary from researcher,  
  summary from writer  
]
```

task("researcher")

SUBAGENT: researcher

```
context = [  
  full web_search results,  
  all 14 sources,  
  notes, drafts, citations  
]  
return: 150-word summary
```

SUBAGENT: writer

```
context = [  
  researcher summary,  
  original user task  
]  
return: polished 200-word report
```

Virtual FS: state files dict

INFO

Files live in graph state, not on disk by default.

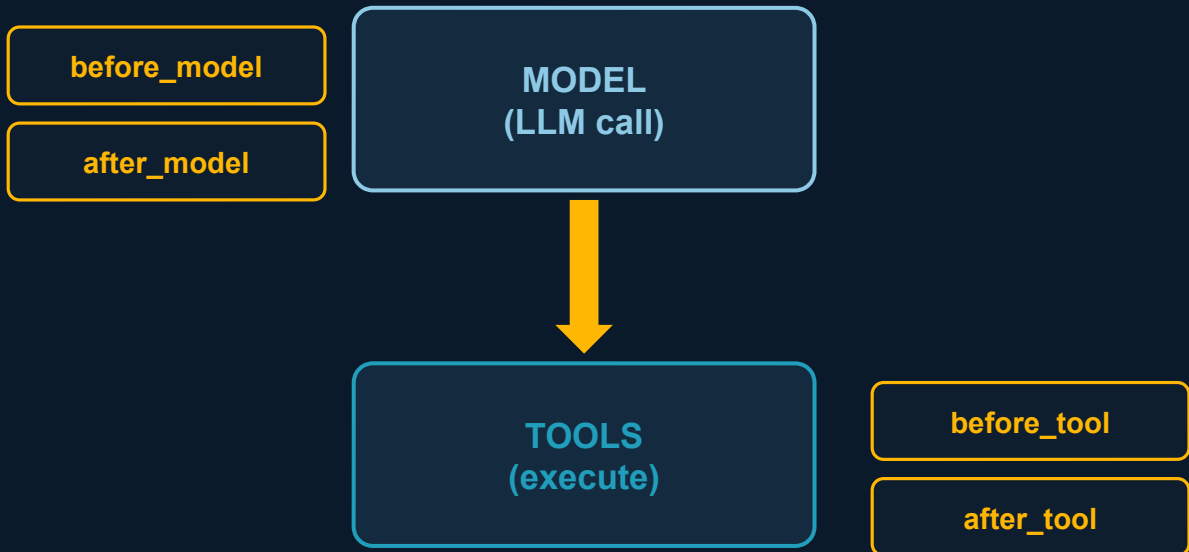
StateBackend keeps everything in state["files"]. Survives across turns in the same thread: never touches disk unless you swap backends

// examples/virtual_fs.py:1-9

```
# Inspect the virtual filesystem after a run:
result = agent.invoke({"messages": "Summarize repo"})
>
> files = result.get("files", {})
> for path, doc in files.items():
>     print(f"{path}: {len(doc.get('content', []))} bytes")

# /repo/src/main.py: 421 bytes
# /repo/README.md: 1804 bytes
```

Middleware: four hook points



INFO

Same middleware system as LangChain 1.0 `create_agent`.

Mix custom middleware with built-ins: Summarization, HumanInTheLoop, Filesystem, SubAgent.

Custom middleware: logging + PII redaction

// examples/middleware.py:1-18

```
from langchain.agents.middleware import AgentMiddleware
from deepagents import create_deep_agent

class LoggerMiddleware(AgentMiddleware):
> def before_model(self, state, runtime):
>     print(f"[model] {len(state['messages'])} msgs in")
>     return state
>
> def after_tool(self, state, runtime, tool_result):
>     print(f"[tool] {tool_result.tool_call_id} -> {len(str(tool_result.content))} chars")
    return state

agent = create_deep_agent(
    model="openai:gpt-4.1",
    middleware=[LoggerMiddleware(), HumanInTheLoopMiddleware(
>     interrupt_on={"execute": True}, # ask before running bash
// note: snippet has 18 lines, card fits ~16
> )],
> )
```

Pluggable backends: 4 flavors

StateBackend

default

Everything in graph state["files"]. Zero disk I/O.
Perfect for short-lived, sandboxed runs.

FilesystemBackend

local disk

Real directory on the host. Persists across runs.
Use when the agent should see/edit your repo directly.

StoreBackend

cross-thread

Lives in a LangGraph Store (Postgres, Redis).
Shared between threads and across sessions.

CompositeBackend

route by path

Route reads/writes to different backends depending on
path prefix.
"/workspace" -> Filesystem, "/memory" -> Store.

CompositeBackend: route by path prefix

// examples/composite_backend.py:1-20

```
from deepagents import create_deep_agent
from deepagents.backends import (
    CompositeBackend, FilesystemBackend, StoreBackend
)
from langgraph.store.memory import InMemoryStore

store = InMemoryStore() # or PostgresStore in prod

backend = CompositeBackend(
    default=FilesystemBackend(root_dir="./workspace"),
    routes={
>     "/memory/": StoreBackend(store=store, namespace=("agent", "kb")),
>     "/scratch/": FilesystemBackend(root_dir="/tmp/scratch"),
> },
> )
```

// note: snippet has 20 lines, card fits ~16

```
agent = create_deep_agent(model=..., backend=backend)
# /workspace/notes.md -> local disk
# /memory/lessons.md -> Postgres, shared across sessions
```

source > # /scratch/tmp.py -> ephemeral tmpfs:kends

interrupt_on: approve tool calls

// examples/hitl.py:1-23

```
from langchain.agents.middleware import HumanInTheLoopMiddleware
from deepagents import create_deep_agent
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.types import Command

agent = create_deep_agent(
    model="openai:gpt-4.1",
    checkpointer=InMemorySaver(),
    > middleware=[HumanInTheLoopMiddleware(
    > interrupt_on={
    > "execute": True, # bash -- always ask
    > "write_file": True, # disk writes -- always ask
    > "task": False, # subagents -- run unattended
    > },
    > ),
    > ),
```

//)note: snippet has 23 lines, card fits ~15

```
cfg = {"configurable": {"thread_id": "user-42"}}
try:
```

source: [https://github.com/langchain-ai/langgraph/blob/main/examples/hitl.py](#)
 agent.invoke({"messages": ["deploy to staging"], cfg))
 except InterruptedError:

stream_mode: tokens, updates, events

// examples/streaming.py:1-18

```
cfg = {"configurable": {"thread_id": "user-42"}}

# 1. Stream model tokens as they arrive
> for token, meta in agent.stream(
>   {"messages": "..."}, cfg,
>   stream_mode="messages",
> ):
    print(token.content, end="", flush=True)

# 2. Stream state updates per node
for chunk in agent.stream(
>   {"messages": "..."}, cfg,
>   stream_mode="updates",
> ):
    print(chunk) # {"model": {...}, "tools": {...}}

// note: snippet has 18 lines, card fits ~16
# 3. Subagent streams are surfaced as
#   {"subagent": {"name": "researcher", "chunk": ...}}
```

INFO stream_mode

values

- "values": full state after each node
- "updates": delta per node (LangGraph-style)
- "messages": token-by-token LLM output
- "events": low-level LangGraph events
- "custom": writer().emit(...) from inside nodes

Tracing and evaluation: works out of the box

INFO

No code changes required.

Set `LANGSMITH_TRACING=true` plus `LANGSMITH_API_KEY` and `LANGSMITH_PROJECT`.

Every deep agent run is traced as a parent run with subagent runs nested underneath.

// examples/langsmith.sh:1-8

```
export LANGSMITH_TRACING=true
export LANGSMITH_API_KEY=lsv2_...
export LANGSMITH_PROJECT=deepagents-evals
```

```
python my_deep_agent.py
# -> all runs visible in smith.langchain.com
# -> subagent runs nested under the parent
# -> token usage, latency, tool errors captured
```

Real example: arXiv research agent

// examples/research_agent.py:1-24

```
from langchain.tools import tool
from deepagents import create_deep_agent
```

```
@tool
```

```
def arxiv_search(query: str, max_results: int = 5) -> str:
```

```
    """Search arXiv for papers matching the query."""
```

```
    import arxiv
```

```
    client = arxiv.Client()
```

```
    results = list(client.results(arxiv.Search(query=query, max_results=max_results)))
```

```
    return "\n\n".join(
        f"{r.title}\n{r.summary[:300]}..." for r in results
    )
```

```
agent = create_deep_agent(
```

```
    model="openai:gpt-4.1",
```

```
    tools=[arxiv_search],
```

// note: snippet has 24 lines, card fits ~16

```
    system_prompt=( "You are a research assistant. Always cite paper titles and arXiv IDs."),
```

```
> subagents=[{
```

```
>     "name": "summarizer",
```

```
source>     "description": "Compresses paper abstracts into a paragraph",
```

```
>     "system_prompt": "You are a precise summarizer."
```

Real example: coding agent, sandboxed

```
// examples/coding_agent.py:1-17
```

```
from deepagents import create_deep_agent
from deepagents.backends import SandboxBackend

> agent = create_deep_agent(
>   model="anthropic:claude-sonnet-4-5",
>   backend=SandboxBackend(
>     provider="daytona",    # or modal / runloop
>     api_key=os.environ["DAYTONA_API_KEY"],
>     image="python:3.12-slim",
>   ),
>   system_prompt=("You are a coding agent. Always run tests after edits. Stop and ask if requirements are
ambiguous."),
> )

agent.invoke({"messages": "Add a /healthz endpoint to the FastAPI app, with tests."})

// note: snippet has 17 lines, card fits ~16
# Daytona/Modal/Runloop execute code in an isolated container;
# the local process never sees a stray rm -rf.
```

1.0-rc and the TypeScript port

INFO

What is new in 1.0-rc

- Full LangChain 1.0 middleware integration
- Stable Send API for parallel subagents
- Pluggable backends marked stable
- Skills: versioning + hot-reload
- CompositeBackend GA
- Note: as of snapshot 2026-06-22, latest stable is 0.6.11 -- 1.0 ships before EOY 2026

// examples/deepagentsjs.ts:1-13

```
// TypeScript analogue (deepagentsjs)
import { createDeepAgent } from "deepagents";
import { tool, z } from "@langchain/core/tools";

const search = tool(
  async ({ q }) => fetch("/api/search?q=" + q).then(r
=> r.text()),
  { name: "search", schema: z.object({ q:
z.string() }) },
);

> const agent = await createDeepAgent({
>   model: "openai:gpt-4.1",
>   tools: [search],
> });
```

When to choose Deep Agents

PROS

- + Batteries included: planning + FS + subagents + HITL + skills in one import
- + Less boilerplate than raw LangGraph, opinionated defaults that match Claude Code
- + Pluggable backends (local / Daytona / Modal / Runloop / LangSmith Store)
- + Skills system: reusable behaviors loaded on-demand
- + Open source (MIT), traceable through

CONS

- 1.0 not yet shipped (0.6.11 latest on snapshot 2026-06-22) -- breaking changes possible
- Opinionated: overriding defaults can be awkward
- Sandbox providers require external SaaS accounts
- Skills ecosystem is nascent, fewer ready-made skills than for Claude Code

INFO

Bridge to Open SWE

Open SWE (next section) is a real coding agent that uses Deep Agents as its harness. All the patterns from this section -- write_todos, subagents,

virtual FS, HITL -- show up unchanged in production at [langchain-ai/open-swe](https://github.com/langchain-ai/open-swe).

4

STAGE 4

Open SWE: async coding agent

Open-source фреймворк LangChain Inc. для построения внутренних кодинг-агентов организации. Reference architecture поверх Deep Agents, pluggable sandboxes, триггеры из Slack/Linear/GitHub, draft PR автоматически. Воспроизводит паттерны Stripe Minions, Ramp Inspect, Coinbase Cloudbot -- но с открытым исходным кодом.

Open SWE: позиционирование

INFO

Не готовый продукт

Стартовый шаблон, не SaaS. Ops-работа: sandbox, модель, триггеры, промпты.

SUCCESS

Colleague, not copilot

Внутренний кодинг-агент, не IDE-assistant.
Slack / Linear / GitHub -> draft PR с тестами.

// meta: open-swe repo:1-12

github.com/langchain-ai/open-swe

stars: ~10k

commits: 971+

license: MIT

language: Python + TypeScript

announce: 08.2025

rewrite: 03.2026

blog.langchain.com/open-swe

INSTALLATION.md

CUSTOMIZATION.md

Архитектура: 7 слоев

Triggers

Slack / Linear / GitHub / Web UI

Validation

prompt + middleware (HITL, approval)

Orchestration

subagents + middleware

Context

AGENTS.md из репозитория

Tools

execute, fetch_url, linear_comment, slack_thread_reply

Sandbox

Modal / Daytona / Runloop / LangSmith

Harness

create_deep_agent (Deep Agents)

```
// open_swe/__init__.py:1-11
```

```
from open_swe.agent import create_agent
from open_swe.middleware import (
    check_message_queue_before_model,
    notify_step_limit_reached,
    open_pr_if_needed,
    ToolErrorMiddleware,
)
from open_swe.sandbox import (
    SandboxBackend,
    ModalBackend, DaytonaBackend,
)
```

create_deep_agent + backend

// examples/minimal_agent.py:1-11

```
from deepagents import create_deep_agent
from open_swe.sandbox import DaytonaBackend

> agent = create_deep_agent(
    model="anthropic:claude-opus-4-6",
    tools=[execute, fetch_url,
          linear_comment,
          slack_thread_reply],
    backend=DaytonaBackend(api_key="..."),
    middleware=[open_pr_if_needed],
)
```

INFO

Один harness

Март 2026: multi-agent
(Manager/Planner/Programmer/Reviewer)

SUCCESS

Upgrade path

Подтягиваешь
улучшения Deep
Agents бесплатно.
Subagents изолируют
контекст, middleware
дают orchestration.

AGENTS.md как system prompt

```
// open_swe/prompts.py:1-9
```

```
from pathlib import Path
```

```
def construct_system_prompt(  
    repo_dir, base_prompt  
):
```

```
    agents_md = Path(repo_dir) / "AGENTS.md"
```

```
    // note: prompt has 9 lines, code file 6  
    extra = agents_md.read_text() if  
        agents_md.exists() else ""
```

```
// AGENTS.md:1-5
```

```
# AGENTS.md (repo root)
```

- use uv, not pip
- run pytest before commit
- never push to main directly

INFO

Convention

Организационный паттерн.
Тот же файл читают Cursor,
Aider, Devin.

WARNING

Подводный

камень

Open SWE доверяет
AGENTS.md. Вредоносный
блок попадает в system
prompt.

Middleware: точки расширения

```
// examples/audit_middleware.py:1-13

from langchain.agents.middleware import (
    AgentMiddleware,
)

> class AuditMiddleware(AgentMiddleware):
>     def after_model(self, state, runtime):
>         runtime.logger.info(
>             f"step={state.get('step')}"
>         )
>         return state

def before_model(self, state, runtime):
    return state # inject reminder
```

INFO

Built-in middleware

- * check_message_queue
- * notify_step_limit
- * open_pr_if_needed
- * ...

SUCCESS

Типичные

кастомы

Approval gate перед PR, cost guard на длину контекста, redaction секретов в логах.

Sandbox-провайдеры

Provider	Setup	Pricing model	Persistent state	Best for
ModalBackend	token_id + token_secret	per-second + GB-s	yes (volume)	Python-first, GPU-доступный
DaytonaBackend	api_key	per-session	yes (volume)	default в README
RunLoopBackend	api_key	per-second	yes (volume)	dev-цикл, snapshot/restart
LangSmithBackend	LANGSMITH_API_KEY	через LangSmith	через LS	уже платите за LangSmith

Sandbox: imports + custom

```
// open_swe/sandbox/__init__.py:1-12
```

```
from open_swe.sandbox import (
    ModalBackend,
    DaytonaBackend,
    RunloopBackend,
    LangSmithBackend,
)

# Modal (Python-first, GPU)
backend = ModalBackend(
    token_id="..."
    token_secret="..."
)
```

```
// examples/my_internal_backend.py:1-16
```

```
# свой backend: devbox-пул
from open_swe.sandbox import (
    SandboxBackend,
)

class MyInternalBackend(
    SandboxBackend
):
    def __init__(self, conn):
        self.conn = conn

>
> def execute(self, cmd):
>     return self._run(cmd)
>
> def read_file(self, p):
>     return self._fetch(p)
// note: snippet has 16 lines, and fits ~15
```


Установка (1/2): шаги 1-2

// INSTALLATION.md: step 1:1-5

```
# 1. prerequisites
python --version # >= 3.11
node --version   # >= 20
uv --version     # или pip
docker --version # для dev
```

// INSTALLATION.md: step 2:1-5

```
# 2. clone + install
git clone https://github.com/
  langchain-ai/open-swe.git
cd open-swe
uv sync # или pip install -e .
```

INFO

He SaaS

Open SWE -- не готовый сервис. После клонирования нужно поднять backend, UI, sandbox. [GitHub App](#).

WARNING

ENV-файл

ср .env.example .env, затем заполните:

- GITHUB_APP_ID
- LANGSMITH_API_KEY
- DAYTONA_API_KEY

Установка (2/2): шаги 3-5

// INSTALLATION.md: steps 3-5:1-13

3. backend (FastAPI)

```
uv run apps/open-swe/main.py
```

```
# -> :8000
```

4. UI (TanStack Start + Vite)

```
cd apps/open-swe-ui
```

```
pnpm install && pnpm dev
```

```
# -> :3000
```

5. smoke-test

```
curl -X POST :8000/webhooks/slack \
```

```
-d '{"text": "@open-swe hello"}'
```

```
# -> {"thread_id": "..."}'
```

S U C C E S S

Готово

Backend :8000

UI :3000

Webhook

8000/webhooks/slack/*

I N F O

Production

Локальный запуск --
только dev. Для prod:
docker compose, k8s,
managed Postgres, Vault,
TLS.

GitHub App: manifest

```
// config/github-app-manifest.yaml:1-13
```

```
# github-app-manifest.yaml
name: open-swe-internal
> url: https://open-swe.example.com
> hook_attributes:
>   url: https://open-swe.example.com/
>   webhooks/github
>   events:
>     - issue_comment
>     - pull_request
>       - pull_request_review
default_permissions:
  contents: write
  pull_requests: write
```

INFO

Создание App

1. github.com/settings/apps/new

WARNING

Permissions

contents:write -- ветки
pull_requests:write -- draft PR
issues:write -- комментарии
metadata:read -- repo info

LangSmith: setup

```
// .env:1-10
```

```
# .env (НЕ коммитить)
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT=https://
  api.smith.langchain.com
LANGSMITH_API_KEY=lsv2_...
LANGSMITH_PROJECT=open-swe-prod
LANGSMITH_SNAPSHOT=true

# для sandbox-прокси:
LANGSMITH_SANDBOX_BACKEND=true
```

INFO

Зачем snapshot

Каждый thread = checkpoint.
Follow-up из всех каналов
полхватывают состояние по

SUCCESS

API keys

- * Personal --
smith.langchain.com
- * Org-level -- shared tracing
- * Service -- для CI / batch
Rotate каждые 90 дней.

Triggers overview

@

Slack

В любом thread канала.
Поддерживает синтаксис
repo:owner/name.

```
@open-swe fix the auth bug  
repo:acme/api
```

#

Linear

В комментарии к issue.
Привязывает thread к
Linear-тикету.

```
@openswe implement the  
acceptance criteria
```

PR

GitHub

В PR-комментарии для
авто-ответа на review-
комментарии.

```
@openswe address  
the review comments
```

Triggers: thread_id routing

```
// open_swe/triggers/router.py:1-11
```

```
# open_swe/triggers/router.py
> def thread_id_for(source, ref, comment_id):
>     # Deterministic thread_id:
>     # все follow-up попадают в один run.
>     if source == "slack":
>         return f"slack:{ref}"
>     if source == "linear":
>         return f"linear:{ref}"
>     if source == "github":
>         return f"github:{ref}:{comment_id}"
>         raise ValueError(source)
```

INFO

Routing

thread_id из (source, ref).
Если run бежит -- новые сообщения ждут в очереди.

WARNING

Concurrency

Один thread = один run.
Параллельность через отдельный thread_id (например, отдельный Slack).

Webhook endpoints

```
// apps/open-swe/webhooks.py:1-15
```

```
# apps/open-swe/webhooks.py
from fastapi import APIRouter, Request
from open_swe.triggers.router import (
    thread_id_for,
)

router = APIRouter()

> @router.post("/webhooks/slack")
> async def slack_webhook(req: Request):
>     payload = await req.json()
>     if "@open-swe" not in payload["text"]:
>         return {"ok": True}
>     await enqueue_run(thread_id_for(
>         "slack", payload["channel"]))
```

SUCCESS

Idempotency

Webhook вычисляет thread_id и проверяет наличие run. Если есть

WARNING

Signature

Все handlers обязаны проверять X-Signature от Slack / Linear / GitHub. Не доверяйте без verify.

Dashboard UI

```
// apps/open-swe-ui/tree.sh:1-13
```

```
# apps/open-swe-ui/  
# TanStack Start + Vite + React 19
```

```
src/routes/  
  __root.tsx  
  index.tsx      # thread list  
  thread.$id.tsx # chat  
  settings.tsx  
src/components/  
  ChatMessage.tsx  
  DiffViewer.tsx  
src/lib/  
  client.ts      # OpenSWEClient
```

INFO

Только UI

Dashboard -- presentation layer. Агентская логика в Python backend. UI через

SUCCESS

Features

- * GitHub OAuth login
- * thread list + filters
- * chat with streaming
- * diff viewer для PR
- * per-user settings

Per-user настройки

```
// open_swe/settings.py:1-15
```

```
# settings schema (per-user)
USER_DEFAULTS = {
>   "model": "anthropic:claude-opus-4-6",
>   "profile": "balanced",
>   "max_steps": 80,
>   "auto_open_pr": False,
>   "require_approval": True,
    "extra_repos": ["acme/internal-tools"],
}

> # team defaults (allowlist)
> TEAM_DEFAULTS = {
    "sandbox_backend": "ModalBackend",
    "allowed_repos": ["acme/*"],
}
```

INFO

User mappings

GitHub login -> Slack user -> Linear user. Один человек получает свой профиль во

WARNING

Precedence

user > team > global. Per-user override работает для всех полей, кроме `allowed_repos` -- это team-level allowlist.

6 точек кастомизации

1

Sandbox provider

```
backend=ModalBackend(...)
```

2

Model

```
init_chat_model("anthropic:...")
```

3

Tools

```
tools=[execute, fetch_url, ...]
```

4

Triggers

```
router.add_route("/my", my_handler)
```

5

System prompt

```
construct_system_prompt(repo_dir, ..)
```

6

Middleware

```
middleware=[AuditLogger(), ...]
```

Three graphs

agent graph

Основной deep-agent: planner + coder + tools + subagents.

START -> plan -> execute -> review -> open_pr -> END

reviewer graph

CI-стиль: lint, type-check, tests, security-scan. Без LLM-агента.

START -> run_ci -> parse -> report -> END

analyzer graph

Анализ ретроспективы: что заняло больше шагов, где loop, где cost spikes.

START -> load_trace -> aggregate -> summarize -> END

Observability

```
// open_swe/observability.py:1-15

# observability.py
from datadog import statsd
> from langchain_core.tracers import (
>     LangChainTracer,
> )

tracer = LangChainTracer(
>     project_name="open-swe-prod",
> )
>

def record_step(thread_id, duration_s):
    statsd.histogram(
        "open_swe.step.duration_s",
        duration_s, tags=[f"t:{thread_id}"],
    )
```

INFO

LangSmith

Trace каждого run: model calls, tool calls, retries.
Replay, debug, manual evaluation

SUCCESS

Datadog

P95 latency, tokens per run, error rate, sandbox spend.
Alerts на cost spikes и длинные loops.

Production: prod-ready

SUCCESS

Infra

- + docker-compose / k8s
- + managed Postgres
- + Vault / sealed-secrets
- + TLS + reverse proxy

INFO

Ops

- + LangSmith prod-project
- + Datadog dashboards + SLO
- + sandbox cost budget
- + audit log всех PR

WARNING

Самый частый пропуск

"Trust the LLM" внутри sandbox. Без надлежащей изоляции (seccomp, network policy, ephemeral FS) агент может сделать `rm -rf` или `curl` внутренних сервисов. Изоляция важнее prompts.

TypeScript: только UI

```
// apps/open-swe-ui/src/lib/client.ts:1-13
```

```
// apps/open-swe-ui/src/lib/client.ts
import { OpenSWEClient } from
"@openswe/client";

const client = new OpenSWEClient({
  langsmithApiKey: process.env.
    LANGSMITH_API_KEY!,
});

await client.invoke({
  threadId: "issue-123",
  prompt: "Fix the bug",
});
```

PROS

- + UI: полностью TS
- + TanStack Start + Vite
- + strict TS config
- + streaming SDK

CONS

- Нет TS для backend
- Агент -- Python only
- Webhook handlers только Py
- SDK -- thin wrapper

Open SWE vs Claude Code / Devin

PROS

- + MIT license, форкайте
- + Pluggable sandbox
- + Triggers: Slack/Linear/GH
- + AGENTS.md convention
- + Subagents + middleware
- + Built on Deep Agents

CONS

- Не finished product
- Sandbox = платные аккаунты
- OAuth setup нужен
- "Trust the LLM" модель
- Prod deployment сложный
- Доки быстро устаревают

VS Claude Code / Devin

Audience	IDE	SaaS	org
License	proprietary	proprietary	MIT
Trigger	manual	Web UI	Slack/Lin/GH
Sandbox	local+cloud	remote	pluggable
Customize	config	no	full src
Runs in	IDE/term	cloud	your infra

Roadmap: 2026-2027

Q2 2026

Stable v1

- > deep-agent harness заморожен
- > API стабилизирован
- > semver commitment

Q3 2026

Multi-tenant

- > per-org quota + billing
- > team-level audit log
- > rbac на sandbox providers

Q4 2026

More triggers

- > Jira / Azure DevOps
- > Sentry для авто-fix багов
- > PagerDuty для incident triage

2027

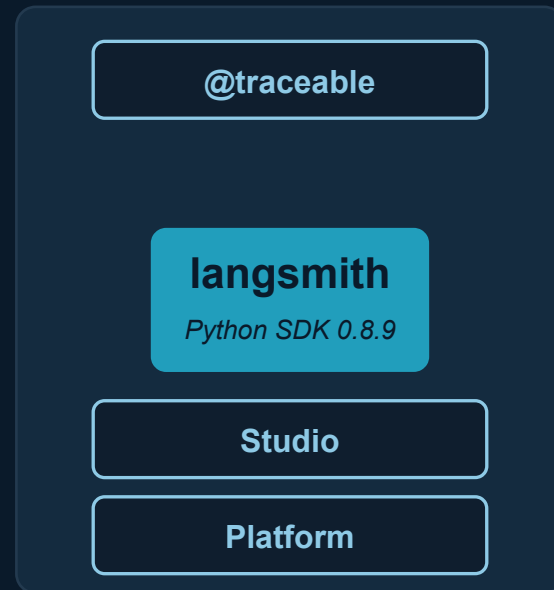
SDK + Marketplace

- > public SDK (Python + TS)
- > plugin marketplace
- > shared subagent library

Экосистема

LangSmith, Studio, deployment

Сервисы вокруг 4 основных ступеней: observability и eval (LangSmith SDK), визуальный дебаг графов (LangGraph Studio), production deployment (LangGraph Platform), self-hosted и TypeScript-клиенты. Все, что превращает прототип в production-grade систему.



observability | visual debug | deploy

Три столпа поверх LangChain/LangGraph

LangSmith -- SaaS-платформа (с self-hosted вариантом) для production-наблюдения и оценки LLM-приложений. Три ключевые возможности покрывают весь цикл: отладка в dev, метрики в prod, оценка качества на датасетах.

Tracing

observability

Каждый вызов Runnable, графа, tool логируется как Run с input/output, latency, token-cost. Дерево вызовов -- в UI, и через SDK.

Datasets

eval sets

Версионизуемые наборы примеров (input + expected output). Используются для off-line eval, regression-тестов, few-shot examples.

Evaluators

quality scoring

LLM-as-judge, heuristic, human -- на выбор. Прогоняются на датасете, результаты привязываются к эксперименту (commit, prompt version).

Cloud: smith.langchain.com | SDK: `langsmith==0.8.9` | Self-hosted v0.9 (changelog 21.01.2025)

pip install и 3 переменные окружения

```
// shell/install_langsmith.sh:1-7
```

```
# Standalone SDK (если не используете langchain)
pip install langsmith
```

```
# С langchain/LangGraph auto-tracing работает
# автоматически при env-переменных -- отдельно
# ставить SDK необязательно.
pip install langchain langgraph
```

```
// shell/env.sh:1-11
```

```
# 1. Tracing on/off (default: true если есть API key)
export LANGSMITH_TRACING=true
```

```
# 2. API key: https://smith.langchain.com/settings
export LANGSMITH_API_KEY="lsv2_pt_..."
```

```
# 3. Project name (default: "default")
export LANGSMITH_PROJECT="my-agent-dev"
```

```
# Опционально: self-hosted endpoint
# export LANGSMITH_ENDPOINT=...
```

INFO

Zero-config для LangChain/LangGraph

Если `LANGSMITH_API_KEY` задан, `chain.invoke/graph.invoke/agent.stream` пишут `trace` автоматически -- без оберток и `monkey-patch`.

source: [docs.smith.langchain.com Observability > Set up tracing](https://docs.smith.langchain.com/Observability%20Set%20up%20tracing)

@traceable -- декоратор для любой функции

```
from langsmith import traceable

# Декоратор превращает функцию в traced Run.
# Все аргументы и return попадают в trace автоматически.
@traceable(name="retrieve_docs", run_type="retriever")
def retrieve(query: str, k: int = 4) -> list[str]:
    return vector_store.similarity_search(query, k=k)

@traceable(name="generate_answer", run_type="chain")
def generate_answer(query: str) -> str:
    docs = retrieve(query)          # nested Run
    context = "\n".join(docs)
    return llm.invoke(f"Q: {query}\nCtx: {context}")

# Дерево: generate_answer -> retrieve_docs -> ChatModel
print(generate_answer("What is LCEL?"))
```

S U C C E S S

Вложенные вызовы автоматически становятся дочерними Run-ами в дереве трассировки.

RunTree -- ручной контроль над деревом

```
from langsmith.run_trees import RunTree

# RunTree -- explicit handle: создается руками, передается
# в код, потом post() отправляет в LangSmith. Нужно когда
# @traceable не подходит (динамич. дерево, не-Python, metadata).
parent = RunTree(
    name="agent_turn",
    run_type="chain",
    inputs={"q": "Who invented Python?"},
    project_name="my-agent-dev",
)
try:
    answer = my_agent(question, run_tree=parent)
    parent.end(outputs={"answer": answer})
except Exception as e:
    parent.end(error=str(e))      # ошибка логируется
    raise
finally:
    parent.post()                # flush в LangSmith
```

// note: snippet has 19 lines, card fits ~17

WARNING

Не забудьте `parent.post()` -- иначе trace не отправится. Дочерние через `parent.create_child()`

Auto-tracing LangChain + метаданные

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_template("Tell a joke about {topic}")
model = ChatOpenAI(model="gpt-4o-mini")
chain = prompt | model

# auto-tracing: invoke/batch/stream автоматически создают Run
# с input, output, token usage, latency.
result = chain.invoke(
    {"topic": "databases"},
    > config={
    >     "run_name": "joke_chain", # имя в UI
    >     "tags": ["prod", "experiment-v3"], # фильтр в UI
    >     "metadata": { # любые поля
    >         "user_id": "u_123",
    >         "request_id": "req_abc",
    < note: snippet has 20 lines, card fits ~17
    > },
    )
```

S U C C E S S

tags и metadata -- способ группировать и фильтровать trace-ы в UI по user/session/feature.

create_dataset -- набор примеров под eval

```
from langsmith import Client

client = Client()

# 1. Создаем датасет (или достаём существующий по имени)
dataset = client.create_dataset(
    dataset_name="rag-qa-eval-v1",
    description="RAG golden set",
)

# 2. Добавляем примеры пачкой: inputs + reference outputs
client.create_examples(
    dataset_id=dataset.id,
    inputs=[{"q": "Что такое LCEL?"}, {"q": "Что такое HITL?"}],
    outputs=[{"a": "LangChain Expression Language"},
              {"a": "Human in the loop opens interrupt"}],
)
// note: snippet has 20 lines, and fits in 16
```

INFO

В UI датасеты можно создавать из CSV/JSONL через web -- API нужен для CI и автообновления.

source: [docs.smith.langchain.com Evaluation > Datasets](https://docs.smith.langchain.com/Evaluation%20Datasets)

run_on_dataset + evaluator-ы

```
from langsmith import Client
from langsmith.schemas import Example, Run

client = Client()

# 1. Target -- что прогоняем (chain, graph, agent, callable)
def target(inputs: dict) -> dict:
    return {"answer": my_chain.invoke(inputs["question"])}

# 2. Evaluator -- scoring function
def answer_match(run: Run, example: Example) -> dict:
    score = 1.0 if example.outputs["answer"] in run.outputs["answer"] else 0.0
    return {"key": "answer_match", "score": score}

# 3. Прогон: target на датасете + scoring
client.run_on_dataset(
    // Note: snippet has 20 lines, card fits ~15
    dataset_name="rag-qa-eval-v1",
    llm_or_chain_factory=target
```

S U C C E S S

Готовые evaluator-ы: llm_as_judge, exact_match, embedding_distance, cot_qa.

Визуальный дебаг графа

Studio -- это desktop/web IDE для LangGraph. Показывает граф как граф, а не как свалку логов. Запускается локально через `langgraph dev`, либо хостится на LangGraph Platform.

GRAPH CANVAS



RUN INSPECTOR

0ms	__start__
+12ms	router
+840ms	agent (LLM call)
+1.2s	tools[search]
+1.4s	agent (LLM call)
+2.1s	END

LangGraph Platform: deploy + invoke

```
# 1. Конфиг: langgraph.json в корне репо
cat > langgraph.json <<'JSON'
{
  "graphs": {"agent": "./agent.py:graph"},
  "env": "./.env"
}
JSON
```

```
# 2. Deploy one-shot
langgraph deploy --name my-agent-prod
```

```
from langgraph_sdk import get_client

client = get_client(url=
    "https://my-agent-prod.us.langgraph.app"
)

# async API: thread + run
thread = await client.threads.create()
run = await client.runs.create(
    thread["thread_id"], "agent", input={"q": "hi"}
)
```

INFO

LangGraph Platform

managed-хостинг для графа: build из langgraph.json, deploy через CLI, получаете HTTPS endpoint + Studio UI + threads + cron + webhooks.

source: langchain-ai.github.io/langgraph/concepts/langgraph_platform/

Self-hosted vs Cloud: что выбрать

LangSmith и LangGraph Platform существуют в двух режимах: managed Cloud (быстрый старт, оплата по usage) и Self-Hosted (ваш K8s/VM, ваши данные остаются внутри периметра). Актуальная self-hosted версия -- v0.9.

Cloud

smith.langchain.com + managed platform

- + Zero-setup: API key и заводите trace-ы
Managed Postgres, Redis, scaling
Studio UI включен в подписку
Latest features сразу
- Данные уходят в чужой VPC (US-region)
Pay-per-trace: cost растет с нагрузкой
Vendor lock на retention policy

Self-Hosted

Docker/Helm chart, v0.9

- + Данные внутри своего VPC/compliance
Flat cost: предсказуемо для prod
Полный контроль над retention
Air-gapped окружения поддерживаются
- Ops: K8s, Postgres, Redis, ClickHouse
Обновления руками (breaking changes)
Studio UI = отдельный пакет
Не все beta-фичи доступны сразу

Рекомендация: Cloud для prototype и команд до 5 инженеров; self-hosted при compliance-требованиях и > 100M trace-ов в месяц.

source: changelog.langchain.com/announcements/langsmith-self-hosted-v0-9

TypeScript SDK и плюсы/минусы

```
import { Client, traceable } from "langsmith";

const client = new Client();

// @traceable decorator -- точно как в Python
const retrieve = traceable(
  async function retrieve(query: string) {
    return vectorStore.similaritySearch(query, 4);
  },
  { name: "retrieve_docs", runType: "retriever" }
);

// run_on_dataset для eval -- тоже есть
await client.runOnDataset(
  "rag-qa-eval-v1", target,
  { evaluators: [answerMatch] }
);
```

PROS

- + Auto-tracing из коробки
- + Eval = CI/CD для промптов
- + Platform = deploy за минуты
- + Self-hosted опция

CONS

- LangSmith SDK 0.x
- Cloud растёт в цене
- Self-hosted требует K8s

STAGE 1

LangChain 1.0

chains, LCEL, agents, retrievers

STAGE 2

LangGraph 1.0

state, nodes, persistence, HITL

STAGE 3

Deep Agents

harness, todos, virtual FS, subagents

STAGE 4

Open SWE

async coding agent, triggers, dashboard

STAGE 5

Экосистема

LangSmith, Studio, deployment

Полная карта эволюции

- 2022-10** LangChain 0.1 -- запуск фреймворка chains
- 2023-10** LangChain 0.1 (stable) -- первый production-ready
- 2024-01** LangGraph 0.1 -- stateful графы
- 2025-08** Deep Agents 0.x -- harness с subagents
- 2025-10** LangChain 1.0 + LangGraph 1.0 (22.10.2025)
- 2025-12** Open SWE 1.0 -- async coding agent

Куда движется стек

Subagents

делегирование доменных задач с собственным state

Virtual filesystem

stateful scratchpad для reasoning-агентов

Human-in-the-loop

production-ready approval flows в LangGraph 1.0+

Open SWE

async coding agent с GitHub-триггерами и dashboard

LangSmith

observability: traces, evals, online monitoring

Главный тренд

От chain-of-prompts к stateful агентам с harness и human-in-the-loop.

LangChain 1.0 -- это stable ядро (chains, LCEL, agents, retrievers). LangGraph 1.0 -- stateful execution layer. Deep Agents -- reasoning harness "из коробки". Open SWE -- продуктовая реализация coding agent. Всё это работает на Python 1.0+ с @traceable из LangSmith.