

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 хотим использовать.])

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/

| -- 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: Slipset 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

source: https://python.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 hooks.
source: <https://langchain.com/docs/python/langchain/middleware>

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.