

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)
```

// note: 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)
// note: snippet has 21 lines, card fits ~18
app = g.compile()
print(app.invoke(State(question="alex")))
```

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

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](https://github.com/langchain-ai/langgraph/concepts/low_level/#why-langgraph)

Узлы: синхронные и асинхронные функции

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

class State(TypedDict):
    out: str

def sync_node(state: State):          # обычная функция
    return {"out": "sync-ok"}

> async def async_node(state: State): # async тоже работает
    await asyncio.sleep(0)
> return {"out": "async-ok"}

g = StateGraph(State)
g.add_node("s", sync_node)
g.add_node("a", async_node)
g.add_edge(START, "s")
g.add_edge("s", "a")
g.add_edge("a", END)
app = g.compile()
```

// g.add_node("s", sync_node) has 2 lines, card fits ~18

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()
// Note: snippet has 20 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)  # does snippet have 26 lines, 49 tokens ~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 = 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/blob/main/concepts/low_level/cycles.py)

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)

# checkpointers - это абстракция для thread persistence
# InMemorySaver - это реализация InMemorySaver
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
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="interrupt")
```

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: композитность 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]
    ...
```

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 персистится 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" } };
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.