LangChain 1.0 & LangGraph 1.0

Навчальний посібник з мультиагентних систем

Від базового агента до Supervisor Pattern.

Покрокові пояснення, архітектурні рішення, приклади коду.

Модуль 5 курсу agents.pro

agents.pro / module5 - Навчальний курс

2025

1. Вступ

Цей документ є навчальним посібником до модуля agents_v1, який демонструє побудову мультиагентних AI-систем за допомогою двох ключових фреймворків від LangChain:

Що таке LangChain 1.0?

LangChain 1.0 (реліз жовтень 2025) - це стабільний фреймворк для розробки додатків на базі великих мовних моделей (LLM). Ключові особливості:
- Стабільний API з гарантією зворотної сумісності до версії 2.0
- Функція create_agent() для швидкого створення агентів
- Callback-система для розширення поведінки агентів
- Інтеграція з LangSmith для моніторингу та трейсингу

Що таке LangGraph 1.0?

LangGraph 1.0 - це бібліотека для побудови складних робочих процесів (workflows) з використанням графів станів (state graphs). Основні можливості:
- StateGraph для оркестрації мультиагентних систем
- Checkpointing (MemorySaver) для збереження стану
- Conditional edges для динамічної маршрутизації
- Підтримка Supervisor Pattern для координації агентів

Чому це важливо?

Мультиагентні системи дозволяють вирішувати складні задачі, розділяючи їх між спеціалізованими агентами. Кожен агент має свою роль, інструменти та контекст. Це забезпечує:
- Модульність: кожен агент відповідає за свою частину
- Масштабованість: легко додавати нових агентів
- Надійність: помилка одного агента не зупиняє систему
- Спостережуваність: кожен крок можна відстежити

2. Архітектура

Модуль agents_v1 побудований як прогресія від простого до складного. Кожен наступний скрипт базується на концепціях попереднього:

Прогресія прикладів

01_basic_agent.py -> create_agent API, реальні tools

02_agent_middleware.py -> Callback handlers для моніторингу

03_rag_agent.py -> StateGraph, Agentic RAG pattern

04_multiagent.py -> Supervisor Pattern, 4 агенти

Загальна схема архітектури

Всі приклади використовують модель gpt-4o-mini через langchain-openai. State описується як TypedDict з Annotated[List, operator.add] для акумулювання повідомлень. Структуровані LLM-відповіді використовують llm.with_structured_output(PydanticModel). Маршрутизація графу реалізована через add_conditional_edges з routing-функцією, що повертає строкові ключі.

Компонент	Бібліотека	Призначення
LLM	langchain-openai	Виклик gpt-4o-mini
Агент	langchain.agents	create_agent()
Граф	langgraph	StateGraph, workflow
Векторне сховище	faiss-cpu	Similarity search
Embeddings	OpenAIEmbeddings	text-embedding-3-small
Checkpointing	MemorySaver	Збереження стану

3. Приклад 01: Базовий агент

Файл: agents_v1/01_basic_agent.py

Перший приклад демонструє створення агента з реальними інструментами через LangChain 1.0 API. Агент здатний отримувати погоду, шукати в інтернеті та виконувати математичні обчислення.

Реальні інструменти (Tools)

На відміну від багатьох навчальних прикладів, тут використовуються реальні API. Кожен tool створюється за допомогою декоратора @tool з langchain_core.tools:

1. get_weather - OpenWeatherMap API

```
@tool
def get_weather(location: str) -> str:
    """Get current weather for a specific location."""
    api_key = os.getenv("OPENWEATHERMAP_API_KEY")
    url = "http://api.openweathermap.org/data/2.5/weather"
    params = {"q": location, "appid": api_key, "units": "metric"}
    response = requests.get(url, params=params, timeout=10)
    data = response.json()
    return f"Weather in {location}: {data['main']['temp']}C"
```

2. web_search - DuckDuckGo

```
@tool
def web_search(query: str) -> str:
    """Search the web using DuckDuckGo."""
    with DDGS() as ddgs:
        results = list(ddgs.text(query, max_results=3))
    formatted = []
    for i, r in enumerate(results, 1):
        formatted.append(f"{i}. {r['title']}\n   {r['body'][:200]}")
    return "\n".join(formatted)
```

3. calculate - numexpr

```
@tool
def calculate(expression: str) -> str:
    """Safe mathematical calculations using numexpr."""
    result = ne.evaluate(expression)
    return f"Result: {result}"
```

Створення агента

Агент створюється через create_agent() - новий API LangChain 1.0, що замінює застарілий AgentExecutor. Виклик агента - через agent.invoke():

Створення та виклик агента:

```
from langchain.agents import create_agent

agent = create_agent(
    model="gpt-4o-mini",
    tools=[get_weather, calculate, web_search],
    system_prompt="You are a helpful AI assistant..."
)

# Виклик агента
result = agent.invoke({
    "messages": [{"role": "user", "content": "What's the weather in Kyiv?"}]
})
```

Invoke Pattern

В LangChain 1.0 агент використовує dict з ключем 'messages'. Результат - також dict з 'messages', де останнє повідомлення містить відповідь:

result = agent.invoke({"messages": [...]})
output = result["messages"][-1].content

4. Приклад 02: Агент з Callbacks

Файл: agents_v1/02_agent_with_middleware.py

Цей приклад показує як розширити поведінку агента через систему callbacks - офіційний механізм LangChain 1.0 для перехоплення подій.

BaseCallbackHandler API

Кожен callback handler наслідує BaseCallbackHandler і перевизначає потрібні методи:
- on_llm_start / on_llm_end - перед/після виклику LLM
- on_tool_start / on_tool_end - перед/після виклику tool
- on_agent_action - при дії агента (вибір tool)

4 типи Callback Handlers

1. LoggingCallback

Детальне логування всіх операцій: виклики LLM, використання tools, таймстемпи та статистика.

```
class LoggingCallback(BaseCallbackHandler):
    def __init__(self):
        self.llm_calls = 0
        self.tool_calls = 0
        self.logs = []

    def on_llm_start(self, serialized, prompts, **kwargs):
        self.llm_calls += 1
        log_entry = {
            "timestamp": datetime.now().isoformat(),
            "event": "llm_start",
            "prompt_length": len(prompts[0]) if prompts else 0
        }
        self.logs.append(log_entry)

    def on_tool_start(self, serialized, input_str, **kwargs):
        self.tool_calls += 1
        tool_name = serialized.get("name", "unknown")
        print(f"TOOL CALL: {tool_name}, Input: {input_str}")
```

2. SecurityCallback

Перехоплює та логує спроби використання небезпечних інструментів (execute_trade, send_notification). В продакшені може блокувати виклики.

```
class SecurityCallback(BaseCallbackHandler):
    def __init__(self):
        self.high_risk_tools = ["execute_trade", "send_notification"]
        self.blocked_calls = 0

    def on_agent_action(self, action: AgentAction, **kwargs):
        if action.tool in self.high_risk_tools:
            self.blocked_calls += 1
            print(f"HIGH-RISK ACTION: {action.tool}")
```

3. TokenCountCallback

Підраховує приблизну кількість використаних токенів та попереджає при наближенні до ліміту. Використовує формулу: ~1 токен = 4 символи.

4. PerformanceCallback

Вимірює час виконання кожного LLM-виклику та tool-виклику. Використовує on_llm_start/on_llm_end та on_tool_start/on_tool_end.

Підключення Callbacks

Передача callbacks в agent.invoke():

```
# Callbacks передаються через config при виклику
result = agent.invoke(
    {"messages": [{"role": "user", "content": query}]},
    config={"callbacks": [logging_cb, security_cb, token_cb, performance_cb]}
)
```

Phoenix Tracing

Приклад також інтегрує Arize Phoenix для трейсингу через OpenTelemetry. Якщо Phoenix сервер доступний на localhost:4317, трейсинг вмикається автоматично:

```
from openinference.instrumentation.langchain import LangChainInstrumentor
from phoenix.otel import register
import phoenix as px

tracer_provider = register()
LangChainInstrumentor(tracer_provider=tracer_provider).instrument()
```

5. Приклад 03: Agentic RAG

Файл: agents_v1/03_rag_agent_langgraph.py

Цей приклад реалізує Agentic RAG (Retrieval-Augmented Generation) з використанням LangGraph StateGraph. На відміну від базового RAG, тут агент динамічно керує стратегією пошуку: перевіряє релевантність знайдених документів і за потреби переписує запит.

Архітектура потоку

RAG Flow

```
User Query
  |-> Retrieve Docs (FAISS)
    |-> Grade Relevance (LLM + structured output)
      |-> relevant?  -> Generate Answer -> END
      |-> irrelevant? -> Rewrite Query -> Retrieve Again
```

RAGState - Визначення стану

RAGState TypedDict:

```
class RAGState(TypedDict):
    question: str                    # Поточне питання
    retrieved_docs: List[Document]   # Знайдені документи
    relevance_grade: str             # "relevant" або "irrelevant"
    rewrite_count: int               # Лічильник перезаписів
    answer: str                      # Фінальна відповідь
    reasoning: Annotated[List[str], add]  # Кроки міркування
```

Knowledge Base (FAISS)

Векторна база знань створюється з Document об'єктів. Кожен документ має page_content та metadata. Для embeddings використовується text-embedding-3-small:

```
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vectorstore = FAISS.from_documents(documents, embeddings)
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 2}  # Топ-2 найрелевантніші
)
```

4 ноди графу

1. retrieve_documents

Використовує FAISS retriever для пошуку релевантних документів за запитом.

2. grade_documents

LLM з structured output оцінює релевантність знайдених документів. Використовує Pydantic модель GradeOutput з полями relevance та reasoning:

```
class GradeOutput(BaseModel):
    relevance: str = Field(description="'relevant' or 'irrelevant'")
    reasoning: str = Field(description="Why this grade")

structured_llm = llm.with_structured_output(GradeOutput)
grade_result = chain.invoke({"question": q, "documents": docs_text})
```

3. rewrite_query

Переписує запит для кращого retrieval якщо документи нерелевантні. Максимум 2 спроби.

4. generate_answer

Генерує фінальну відповідь на основі знайдених документів та контексту.

Побудова графу

Побудова StateGraph:

```
workflow = StateGraph(RAGState)

# Ноди
workflow.add_node("retrieve", retrieve_documents)
workflow.add_node("grade", grade_documents)
workflow.add_node("rewrite", rewrite_query)
workflow.add_node("generate", generate_answer)

# Ребра
workflow.set_entry_point("retrieve")
workflow.add_edge("retrieve", "grade")
workflow.add_conditional_edges("grade", decide_next_step, {
    "generate": "generate",
    "rewrite": "rewrite"
})
workflow.add_edge("rewrite", "retrieve")
workflow.add_edge("generate", END)

# Компіляція з checkpointing
checkpointer = MemorySaver()
app = workflow.compile(checkpointer=checkpointer)
```

Checkpointing з MemorySaver

MemorySaver зберігає стан виконання графу. Кожна сесія ідентифікується через thread_id, що дозволяє вести окремі розмови:

```
config = {"configurable": {"thread_id": "session_1"}}
result = agent.invoke(initial_state, config)
```

6. Приклад 04: Мультиагентна система

Файл: agents_v1/04_multiagent_langgraph.py

Найскладніший приклад реалізує Supervisor Pattern - архітектуру де центральний агент (Supervisor) координує команду спеціалізованих агентів.

Supervisor Pattern

Supervisor отримує запит користувача і послідовно делегує роботу спеціалістам:

1. Supervisor оцінює поточний стан
2. Вирішує який агент має працювати далі
3. Спеціаліст виконує роботу
4. Результат повертається до Supervisor
5. Цикл повторюється поки не буде FINISH

MultiAgentState

Спільний state для всіх агентів:

```
class MultiAgentState(TypedDict):
    messages: Annotated[List, operator.add]  # Комунікація
    question: str                            # Питання користувача
    current_agent: str                       # Активний агент
    retrieved_docs: List[Document]           # Документи (Researcher)
    analysis: str                            # Аналіз (Analyzer)
    final_answer: str                        # Відповідь (Synthesizer)
    supervisor_decision: str                 # Рішення supervisor
    iteration_count: int                     # Лічильник ітерацій
```

4 спеціалізовані агенти

Агент	Роль	Операції
Supervisor	Координатор	Делегування, контроль
Researcher	Пошук	RAG retrieval, оцінка якості
Analyzer	Аналіз	Витяг insights, структуризація
Synthesizer	Синтез	Фінальна відповідь

Structured Output для рішень

Supervisor використовує Pydantic для гарантованого формату рішень:

```
class SupervisorDecision(BaseModel):
    next_agent: Literal["researcher","analyzer","synthesizer","FINISH"]
    reasoning: str

structured_llm = llm.with_structured_output(SupervisorDecision)
decision = structured_llm.invoke(messages)
```

Побудова мультиагентного графу

Граф з циклом через supervisor:

```
workflow = StateGraph(MultiAgentState)

# Ноди
workflow.add_node("supervisor", supervisor_node)
workflow.add_node("researcher", researcher_node)
workflow.add_node("analyzer", analyzer_node)
workflow.add_node("synthesizer", synthesizer_node)

# Entry point
workflow.set_entry_point("supervisor")

# Conditional routing від supervisor
workflow.add_conditional_edges("supervisor", route_after_supervisor, {
    "researcher": "researcher",
    "analyzer": "analyzer",
    "synthesizer": "synthesizer",
    "end": END
})

# Кожен агент повертається до supervisor
workflow.add_edge("researcher", "supervisor")
workflow.add_edge("analyzer", "supervisor")
workflow.add_edge("synthesizer", "supervisor")
```

7. Ключові патерни та порівняння

Коли використовувати який підхід?

Підхід	Коли використовувати	Складність
create_agent	Прості задачі з tools	Низька
Callbacks	Моніторинг, безпека, логування	Середня
StateGraph RAG	Адаптивний пошук документів	Середня
Supervisor	Складні мультиагентні задачі	Висока

Ключові патерни LangChain 1.0

- Всі LLM виклики через gpt-4o-mini (langchain-openai)
- State як TypedDict з Annotated[List, operator.add] для акумулювання
- Structured output через llm.with_structured_output(PydanticModel)
- Routing через add_conditional_edges з функцією що повертає str
- Checkpointing через MemorySaver (dev) або PostgresSaver (prod)
- Callbacks для cross-cutting concerns (logging, security, metrics)

Типові помилки

- Використання AgentExecutor замість create_agent (deprecated)
- Забути thread_id при checkpointing - стани змішуються
- Не обмежувати кількість rewrites/iterations - нескінченний цикл
- Ігнорування structured output - нестабільний парсинг відповідей

8. Налаштування та запуск

Швидкий старт

```
cd agents_v1
python3 -m venv venv && source venv/bin/activate
pip install -r requirements.txt
cp .env.example .env  # заповніть API ключі
python 01_basic_agent.py
```

Необхідні змінні оточення

Змінна	Обов'язкова	Джерело
OPENAI_API_KEY	Так	platform.openai.com
OPENWEATHERMAP_API_KEY	Для 01_*	openweathermap.org/api
LANGCHAIN_TRACING_V2	Ні	true для LangSmith
LANGCHAIN_API_KEY	Ні	smith.langchain.com

Ключові залежності

requirements.txt (основні):

```
langchain>=1.0
langgraph>=1.0
langchain-openai
faiss-cpu
yfinance
duckduckgo-search
numexpr
arize-phoenix
openinference-instrumentation-langchain
```

Python версія

Рекомендовано Python 3.10-3.13. Уникайте Python 3.14 через проблеми сумісності з Pydantic.