생성형 AI 기반 RAG 개발자 과정

LangChain 소개
LCEL · PromptTemplate · Chain

Session 9 / 33 — Day 2

raw API 반복 코드를 단일 파이프라인으로 — LangChain 표준 추상화

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

이번 시간을 마치면

개념

- ✅ **LangChain 의 필요성** — raw API 대비 이점
- ✅ **버전 변천사·패키지 계층** — 0.1→1.0, core/partner/community 분리 근거
- ✅ **ChatModel · Message** 4종 타입
- ✅ **PromptTemplate · ChatPromptTemplate** — 변수 슬롯 분리
- ✅ **OutputParser** — Str · Pydantic · 구조화 출력

LCEL 체이닝

- ✅ **LCEL** (`prompt | model | parser`) 의 의미 — Unix pipe 모델
- ✅ `RunnableLambda / RunnablePassthrough / RunnableParallel` 로 분기·병렬

> 📂 베이스 소스: `tutorial-genai/2.langchain/1~4.*`

PART A

PART A
LangChain 의 필요성

raw API 와의 비교 — 약 10분

Session 4 코드 vs LangChain 코드

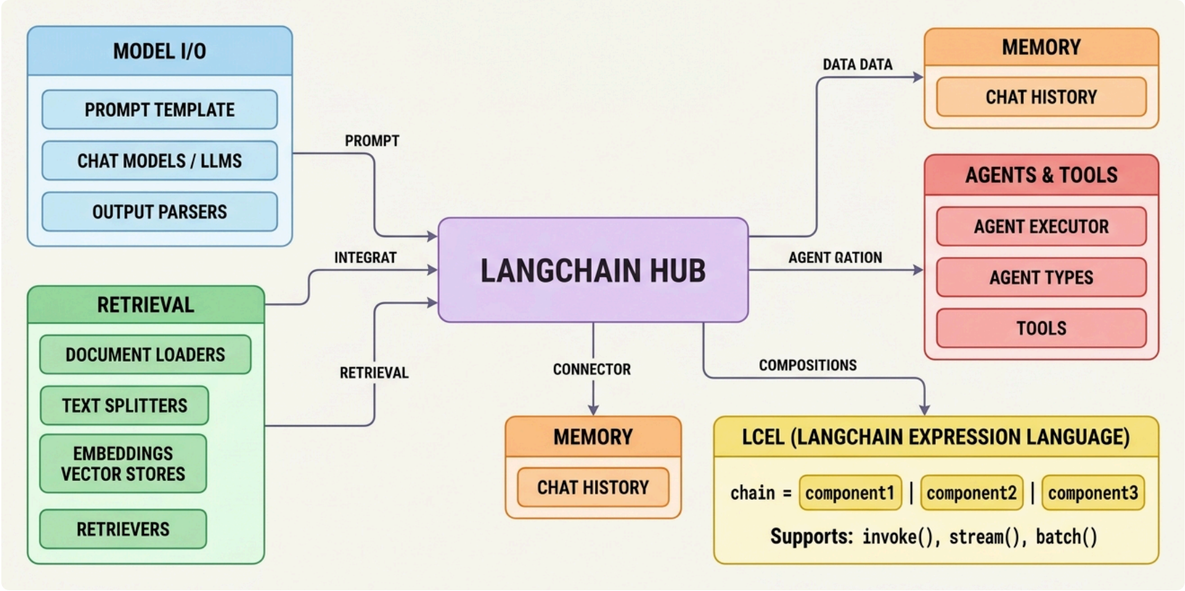
항목	raw API (Session 4)	LangChain (오늘)
모델 교체	OpenAI → Anthropic 시 코드 절반 재작성	`ChatOpenAI()` → `ChatAnthropic()` 한 줄 교체
프롬프트 재사용	f-string 으로 짜깁기	`PromptTemplate` 으로 일관 관리
응답 파싱	매번 `r.choices[0].message.content`	`StrOutputParser` 가 자동
멀티 LLM 조합	for 루프로 직접	`RunnableParallel` 한 줄
구조화 출력	JSON 문자열 → `json.loads` → try/except	`PydanticOutputParser` + validation

LangChain 이 해결하는 5가지

1. **모델 추상화** — 같은 인터페이스로 OpenAI/Anthropic/Gemini/로컬
2. **프롬프트 관리** — 템플릿·변수·few-shot 일관 처리
3. **출력 파싱** — 구조화·검증·재시도 자동화
4. **체이닝** — 여러 LLM/도구를 한 흐름으로 (LCEL)
5. **메모리·RAG·Agent** — 다음 회차들의 모든 추상화

> 🎯 LangChain 은 **LLM 앱 개발의 표준 컴포넌트 집합**이다.

4개 구성요소를 LCEL 로 조립



> LangChain 개요 · 모델 I/O·Retrieval·Memory·Agents/Tools를 LCEL(Runnable 파이프)로 조립

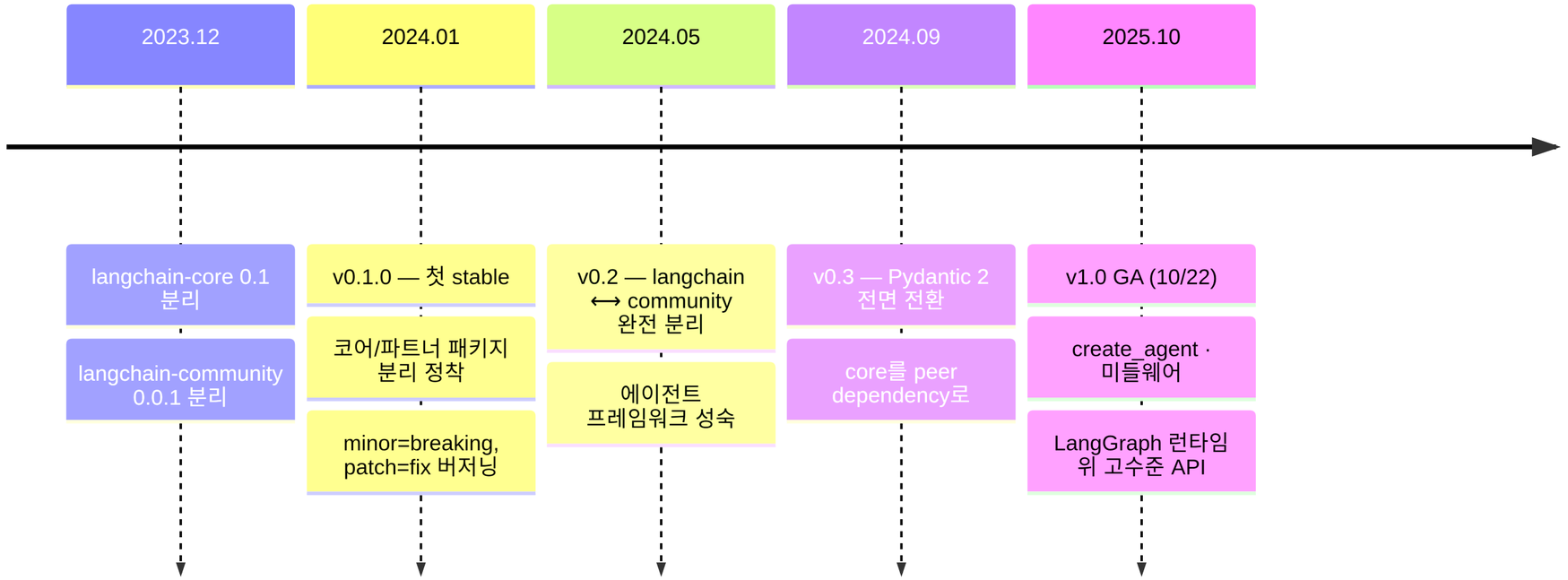
LangChain 구성요소 4분류

- **모델 I/O** — ChatModel · PromptTemplate · OutputParser (오늘 다룰 핵심)
- **Retrieval** — 문서 로더 · 청킹 · 벡터스토어 · Retriever (Day 2 후반)
- **Memory** — 대화 history 관리 (MessagesPlaceholder)
- **Agents / Tools** — 도구 호출 · 자율 워크플로우 (Day 4)

> 🎯 LangChain 의 역할 — **모델 호출·검색·메모리·에이전트를 provider 무관 표준 인터페이스로 묶어 LCEL(Runnable 파이프)로 조립**한다.

0.0.x 실험기 → 1.0 안정기

공식 릴리스 이력. 강의 코드는 **1.0 기준**으로 작성한다.

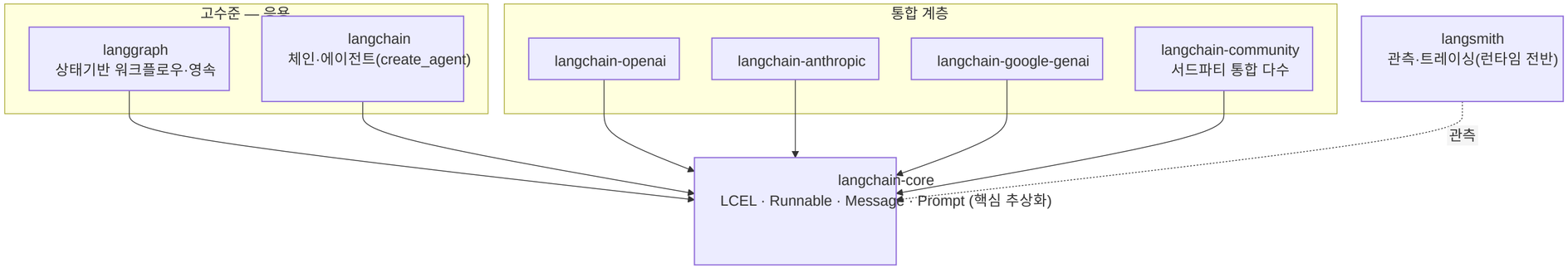


메이저 버전이 뜻하는 것

메이저별 한 줄 요지

- **0.1.0 (2024.01)** — "첫 stable". 이전까지 0.0.x 실험기. 코어·파트너 패키지 분리가 이때 정착.
- **0.2 (2024.05)** — `langchain` 과 `langchain-community` 를 완전히 분리해 안정성·보안 강화.
- **0.3 (2024.09)** — Pydantic 1 EOL(2024.06)에 맞춰 **Pydantic 2** 전면 전환. 구 메모리 클래스 deprecated.
- **1.0 (2025.10.22)** — `create_agent` 가 표준 에이전트 진입점. LangChain = LangGraph 런타임 위의 고수준 API.

계층으로 분리된 패키지 구조



> 모든 패키지가 `langchain-core` 에 의존 · 무거운 통합은 떼어내 독립 릴리스

계층 분리의 이유

분리 근거 (공식)

- **langchain-core** — LCEL·Runnable·Message 등 핵심 추상화만 포함. 모든 패키지가 여기에 의존. 가볍고 안정적이어서 변경이 드물다.
- **partner 패키지** (`langchain-openai` · `langchain-anthropic` 등) — provider SDK 출시 주기가 제각각이라 **독립 버전**으로 릴리스해 한쪽 업데이트가 다른 통합을 깨지 않게 한다.
- **langchain-community** — 방대한 서드파티 통합을 분리해 의존성 비대를 막고 테스트·버전 관리를 단순화한다 (0.2에서 `langchain` 과 완전 분리).
- **langgraph** — 상태기반·영속 워크플로우 런타임. 1.0부터 `langchain` 이 이 위의 고수준 API.
- **langsmith** — 관측·트레이싱(별도 SaaS 연동).

설치 (오늘 사용분)

```
pip install langchain langchain-openai langchain-anthropic langchain-community
```

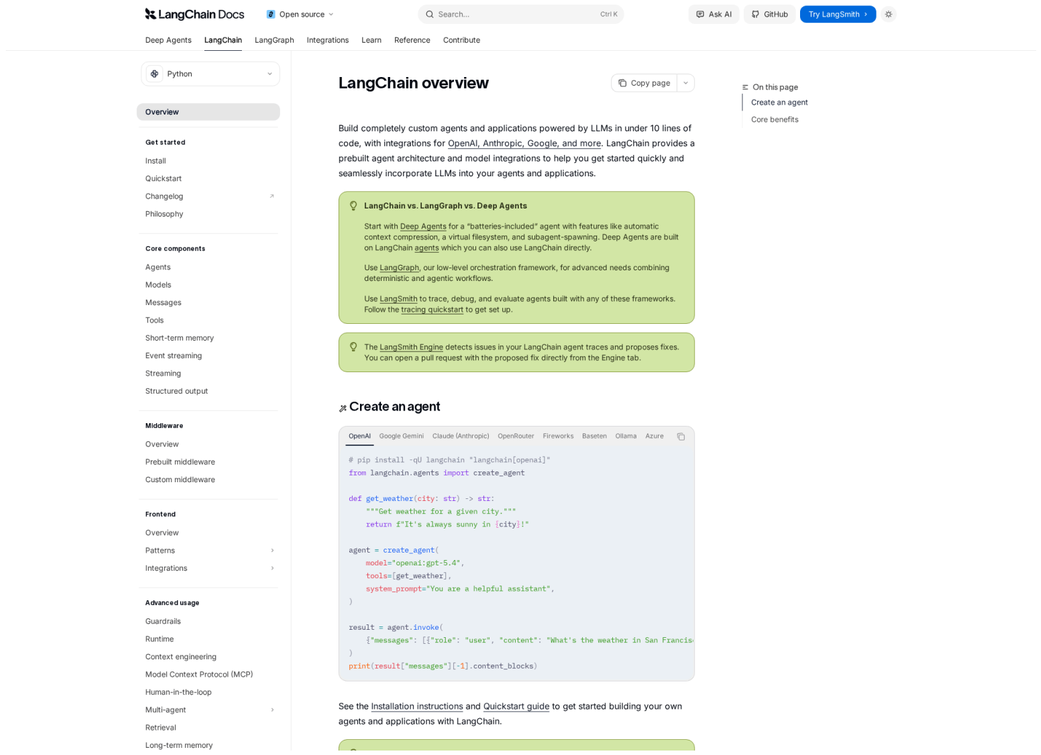
> 💡 핵심 한 줄: **무거운 통합을 core에서 떼어내 독립 릴리스** → 안정성·의존성 관리·provider별 속도 차이를 동시에 해결한 게 패키지 분리의 이유.

docs.langchain.com 주요 페이지

자주 참조하는 페이지 5개

페이지	용도
Introduction	전체 그림, Tutorial 진입점
Concepts (→)	핵심 추상화 한 페이지 정리
LCEL (→)	LangChain Expression Language 문법
Integrations	환경 (모델·벡터DB·도구) 검색 시
How-to guides	작업 단위 레시피 (예: "structured output 받기")

docs.langchain.com — Introduction 화면



PART B

PART B
첫 LangChain 호출

ChatOpenAI · invoke · messages — 약 10분

단일 메서드 호출 패턴

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.7)

# ① 가장 단순한 호출 — 문자열만
result = llm.invoke("한국어로 자기소개해줘")
print(result.content)

# ② messages 리스트 (system + user)
result = llm.invoke([
```

Runnable 실행 메서드

메서드	의미
`invoke(...)`	1개 입력 → 1개 출력 (동기)
`batch([...])`	N개 입력 병렬 처리
`stream(...)`	청크 단위 iteration (SSE 처럼)
`ainvoke / abatch / astream`	비동기 (await) 버전

> 💡 **모델 교체**: `ChatOpenAI` → `ChatAnthropic(model='claude-haiku-4-5')` — 한 줄로 완료.

LangChain Message — 4 종류

```
from langchain_core.messages import (
    SystemMessage,    # 페르소나·규칙
    HumanMessage,     # 사용자 입력
    AIMessage,        # 모델 응답 (history 재사용 시)
    ToolMessage,      # 도구 실행 결과 (Day 4 Agent)
)

messages = [
    SystemMessage("Python 코딩 전문가. 한국어 답변."),
    HumanMessage("dict comprehension 예시?"),
    AIMessage("{k: v for k, v in items}"),       # 과거 응답
    HumanMessage("이걸 set 으로?"),
]

result = llm.invoke(messages)
print(result.content)
print(type(result))      # → <class 'langchain_core.messages.ai.AIMessage'>
print(result.content)    # → 응답 텍스트
print(result.usage_metadata)  # → {'input_tokens': ..., 'output_tokens': ...}
```

Session 4 와의 차이

- Session 4: `{"role": "user", "content": "..."}` (dict)
- LangChain: `HumanMessage("...")` (객체) — 타입 힌트 + IDE 자동완성

PART C

PART C
PromptTemplate — 프롬프트 재사용

f-string 대비 이점 — 약 15분

`2.prompts/1.1_template.py` — 가장 기본형

```
from langchain_core.prompts import PromptTemplate

# ① 템플릿 정의 — {변수} 위치에 나중에 값 주입
prompt = PromptTemplate.from_template(
    "당신은 작명 컨설턴트. {product} 를 만드는 회사 이름을 하나 제안하세요."
)

# ② 값 주입
filled = prompt.format(product="로봇 장난감")
print(filled)
# → 당신은 작명 컨설턴트. 로봇 장난감 를 만드는 회사 이름을 하나 제안하세요.

# ③ 반복 활용 — 같은 템플릿, 다른 입력
for p in ["전기 자전거", "친환경 포장재", "어학 학습 앱"]:
    print(prompt.format(product=p))
```

f-string 대비 이점

- **변수 명시**: 슬롯 목록을 `prompt.input_variables` 로 확인 가능
- **검증**: 누락 시 명확한 메시지로 에러 (`KeyError` 가 아님)
- **체이닝**: `prompt | model | parser` 패턴에서 즉시 사용 (다음 PART)
- **partial**: 일부 변수만 먼저 바인딩 (`.partial(product='기본')`)

챗 모델용 — `from_messages`

```
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "당신은 {role} 전문가. {tone} 톤으로 답하세요."),
    ("user",   "{question}"),
])

# 3개 변수를 한 번에 채워서 messages 리스트 생성
messages = prompt.format_messages(
    role="한국 부동산",
    tone="신중하고 보수적인",
    question="강남 30평 아파트 전세 시세는?",
)

for m in messages:
    print(f"[{m.type}] {m.content}")
# [system] 당신은 한국 부동산 전문가. 신중하고 보수적인 톤으로 답하세요.
# [user]   강남 30평 아파트 전세 시세는?
```

history 슬롯을 사전 정의

MessagesPlaceholder — history 슬롯

```
from langchain_core.prompts import MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages([
    ("system", "Python 페어 프로그래머."),
    MessagesPlaceholder("history"),     # ← 과거 대화 자리
    ("user", "{question}"),
])
prompt.format_messages(history=past_messages, question="이번엔?")
```

> 🎯 **history 관리**가 한 줄로 처리된다. Session 5 의 30줄 코드를 대체.

PART D

PART D
OutputParser — 응답을 코드 친화적으로

문자열 → 구조화 객체 — 약 12분

3.structured_output/1.str_output_parser.py

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

llm = ChatOpenAI(model="gpt-4o-mini")

prompt = ChatPromptTemplate.from_template(
    "{product} 를 만드는 회사 이름을 1개 추천."
)
parser = StrOutputParser()

# 체이닝 — pipe 연산자!
chain = prompt | llm | parser

result = chain.invoke({"product": "전기 자전거"})
```

parser 미사용 시

```
# parser 없이
chain = prompt | llm
result = chain.invoke({"product": "전기 자전거"})
print(type(result))       # <class 'AIMessage'>  ← 객체
print(result.content)     # 실제 문자열은 .content 로
```

> 💡 **체인 끝에 parser 를 두면** 이후 코드에서 응답을 일관되게 문자열로 다룰 수 있다.

JSON 보장 + 타입 검증

```
from pydantic import BaseModel, Field
from langchain_core.output_parsers import PydanticOutputParser

class CompanyName(BaseModel):
    name: str = Field(description="회사명")
    reason: str = Field(description="이름의 의미")
    confidence: float = Field(description="0~1, 추천 강도")

parser = PydanticOutputParser(pydantic_object=CompanyName)

prompt = ChatPromptTemplate.from_messages([
    ("system", "회사명 작명 컨설턴트. {format_instructions}"),
    ("user", "{product} 를 만드는 회사 이름 1개."),
]).partial(format_instructions=parser.get_format_instructions())

chain = prompt | llm | parser

result = chain.invoke({"product": "친환경 운동화"})
print(result)
# CompanyName(name='GreenStride', reason='...', confidence=0.85)
print(result.name)         # 'GreenStride'
```

핵심 — `format_instructions`

- Pydantic 모델의 JSON 스키마를 프롬프트에 자동 삽입
- 모델이 JSON 으로 응답 → parser 가 Pydantic 객체로 검증

OpenAI / Anthropic 의 native 구조화 출력

새 버전 (LangChain 0.2+) — 가장 권장

```
from pydantic import BaseModel, Field
from langchain_openai import ChatOpenAI

class CompanyName(BaseModel):
    name: str = Field(description="회사명")
    reason: str
    confidence: float = Field(ge=0, le=1)

# 모델 자체에 스키마 묶기 — Function Calling 사용
llm = ChatOpenAI(model="gpt-4o-mini").with_structured_output(CompanyName)
```

3가지 옵션 비교

방식	신뢰성	권장도
문자열 응답 → `json.loads` (raw)	낮음 (LLM 이 ```` ```json ```` 감싸기 등)	❌
`PydanticOutputParser` (프롬프트 기반)	중간	⭐⭐
`with_structured_output` (Function Calling)	높음	⭐⭐⭐⭐⭐

> 🎯 **2026 신규 코드는 `with_structured_output` 기본**. Function Calling 이 모델 차원에서 JSON 형식을 강제하기 때문.

PART E

PART E
LCEL — 체이닝 패턴

> `prompt | model | parser` 의 본질 — 약 12분

Unix pipe 와 동일한 합성 모델

파이프를 따라 흐르는 타입

```
{"product": ...}  →  PromptValue  →  AIMessage  →  str
   (dict 입력)       ChatPrompt-      ChatOpenAI    StrOutput-
                      Template                       Parser
```

- 각 Runnable 의 출력 타입이 다음 Runnable 의 입력 타입과 자동으로 맞물린다 — 중간 변환 코드 불필요.
- 같은 체인을 `invoke` (1건) · `stream` (청크) · `batch` (N건 병렬) 로 호출.

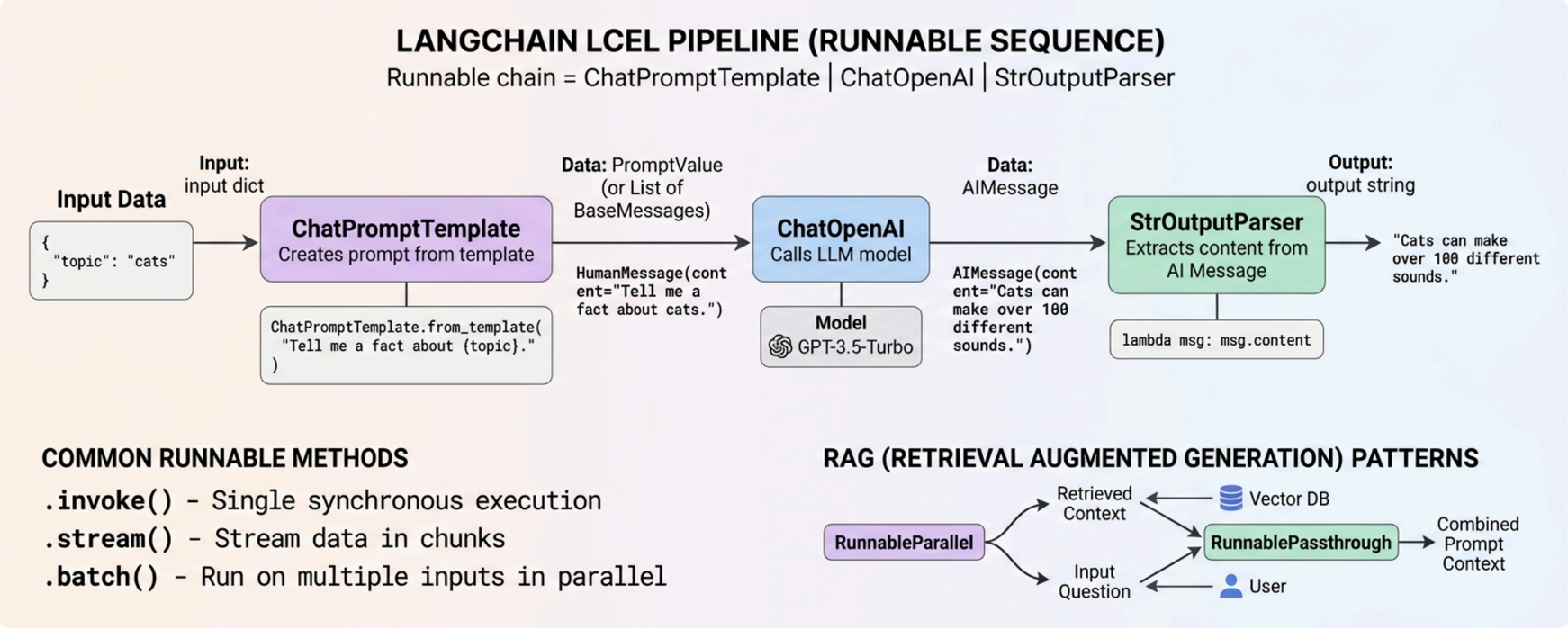
Unix shell

```
cat file.txt | grep error | wc -l
#  data → filter → count
```

LangChain LCEL

```
chain = prompt | model | parser
# input → prompt 채움 → 모델 호출 → 파싱
chain.invoke({"product": "스마트워치"})
```

LCEL — prompt | model | parser



LCEL · ChatPromptTemplate | ChatOpenAI | StrOutputParser · dict→PromptValue→AIMessage→str, invoke/stream/batch

모든 컴포넌트가 동일한 Runnable

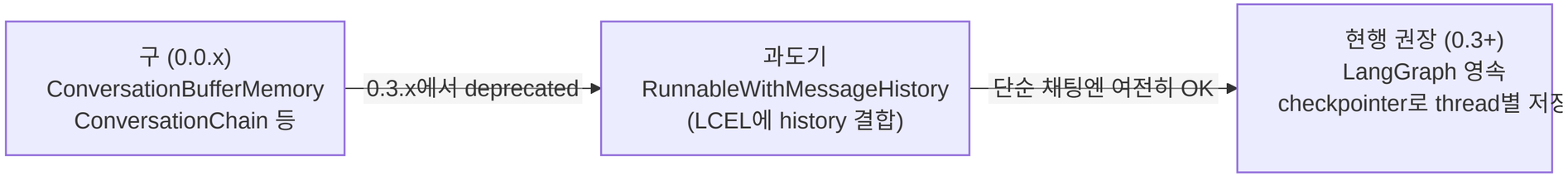
Runnable 이 보장하는 것

1. **공통 인터페이스** — 모든 컴포넌트가 `.invoke / .batch / .stream` 지원
2. **자동 비동기** — 동일 체인을 `.ainvoke()` 로 비동기 실행
3. **자동 병렬화** — `batch()` 시 입력별 자동 병렬
4. **타입 추론** — 체인 입력·출력 타입을 IDE 가 추론

일반 패턴 5종

```
prompt | llm | parser                        # 기본
prompt | llm.bind_tools([t1, t2])            # 도구 사용 (Day 4)
prompt | llm.with_structured_output(Model)   # 구조화 출력
prompt | llm | parser | RunnableLambda(fn)   # 후처리
{"a": chainA, "b": chainB} | combine_chain   # 병렬 후 합치기
```

대화 기억 — 현행 권장 방식



세 단계 정리 (공식 마이그레이션 문서 기준)

- **구 메모리 클래스** — `ConversationBufferMemory` 등. **0.3.x에서 공식 deprecated**.
- `RunnableWithMessageHistory` — LCEL 체인에 history를 결합하는 방식. **deprecate 예정 없음**, 단순 챗봇에는 충분. 다만 API 직관성이 떨어진다는 평이 많았다.
- **LangGraph 영속(권장)** — 0.3부터 공식 권장. `InMemorySaver` · `SqliteSaver` 등 **checkpointer**로 thread별 대화를 저장·복원. 멀티 스레드·고급 기능 지원.

> 🎯 새 코드 기준: 간단하면 `RunnableWithMessageHistory`, 진지한 메모리/멀티턴이면 **LangGraph checkpointer**. 구 `...Memory` 클래스는 신규 채택하지 말 것.

4.chaining/2.1_runnablelambda_instruct.py

```
from langchain_core.runnables import RunnableLambda

def to_uppercase(text: str) → str:
    return text.upper()

def count_words(text: str) → dict:
    return {"text": text, "word_count": len(text.split())}

chain = (
    prompt
    | llm
    | StrOutputParser()
    | RunnableLambda(to_uppercase)       # ← 후처리 1
    | RunnableLambda(count_words)        # ← 후처리 2
)

result = chain.invoke({"product": "스마트워치"})
print(result)
# {'text': 'TIMEMASTER PRO IS A SLEEK SMARTWATCH …', 'word_count': 12}
```

임의의 함수 삽입 가능

- 외부 API 호출 · DB 저장 · 로깅·메트릭

> 💡 람다 함수는 **순수 함수**를 권장한다 (입출력만, 부작용 없음). batch·stream·async 호환을 보장하기 위함.

4.chaining/3.1_runnablepassthrough_*.py

```
from langchain_core.runnables import RunnablePassthrough, RunnableParallel

# 1. Passthrough — 원본을 그대로 통과시키며 추가 키를 만든다
chain = (
    RunnablePassthrough.assign(
        word_count=lambda x: len(x["text"].split())
    )
)
chain.invoke({"text": "hello world"})
# → {'text': 'hello world', 'word_count': 2}
```

동일 입력으로 여러 체인 병렬 실행

```
from langchain_core.runnables import RunnableParallel

# Parallel — 같은 입력으로 여러 체인을 병렬 실행
analysis_chain = (
    RunnableParallel({
        "summary":    prompt_summary    | llm | StrOutputParser(),
        "translation":prompt_translate  | llm | StrOutputParser(),
        "sentiment":  prompt_sentiment  | llm | StrOutputParser(),
    })
)

result = analysis_chain.invoke({"text": "오늘 회의는 매우 생산적이었습니다."})
# → {
#     'summary':     '회의가 생산적이었음',
#     'translation': "Today's meeting was very productive.",
#     'sentiment':   'positive (0.92)',
#   }
```

자동 병렬화 효과

- 3개 LLM 호출이 **동시** 실행 (전체 소요 ≈ 가장 느린 1개)
- 직렬 대비 약 **3배 단축**

핵심 6가지

1. ✅ LangChain = **모델·프롬프트·파서·체인**의 표준 추상화 (provider 무관 인터페이스)
2. ✅ **버전 좌표** — 0.1=첫 stable · 0.3=Pydantic 2 · 1.0=`create_agent` 표준
3. ✅ **패키지 계층** — `langchain-core` 위에 partner·community·langgraph (독립 릴리스로 분리)
4. ✅ raw API 반복 코드를 **LCEL 단일 표현식**으로 — `prompt | model | parser`
5. ✅ **OutputParser** — Str · Pydantic · `with_structured_output` 으로 구조화
6. ✅ **Memory** — 구 `...Memory` 는 deprecated, 현행은 LangGraph checkpointer 권장