

LangChain & LlamaIndex

Frameworks for building RAG pipelines, agents, and LLM applications with less boilerplate.

Last reviewed 28 Sep 2026 ·

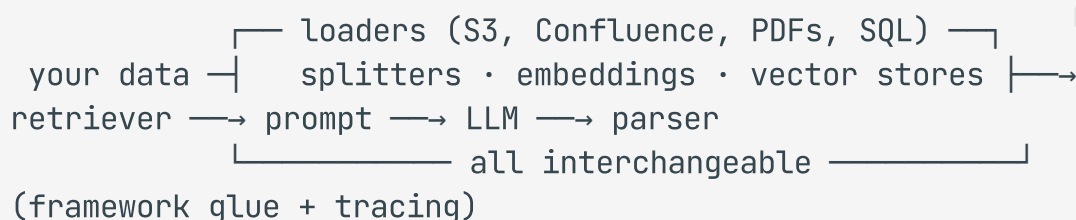
Prerequisites: [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) · [AI Agents](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/)

Related: [AI Observability](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) · [Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

Challenge: LLM applications repeat the same integration work: loading documents, splitting them into chunks, embedding, writing to a vector store, retrieving, formatting prompts, calling the model, parsing output, and tracing. Building this from scratch for every project is slow, and changing model provider or vector database requires rewriting it.

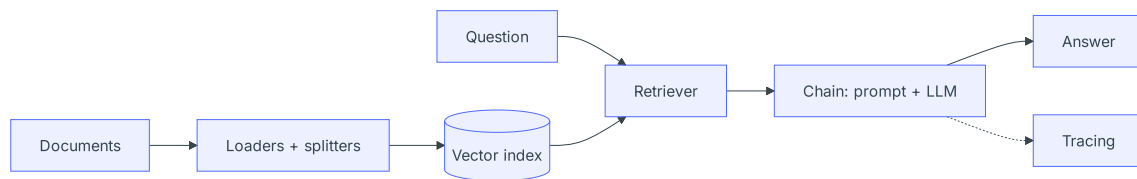
Solution: these frameworks provide ready-made, interchangeable building blocks. - **LangChain** focuses on composing steps — models, prompts, tools, and retrievers — into chains and agents (its LangGraph library handles stateful, multi-step agents). - **LlamaIndex** focuses on *data*: ingesting documents, building indexes, and advanced retrieval for question-answering over your content.



```
graph LR
    subgraph "your data"
        loaders["loaders (S3, Confluence, PDFs, SQL)"]
        splitters["splitters"]
        embeddings["embeddings"]
        vector_stores["vector stores"]
    end
    retriever["retriever"] --> prompt["prompt"]
    prompt --> LLM["LLM"]
    LLM --> parser["parser"]
    retriever -.-> loaders
    retriever -.-> splitters
    retriever -.-> embeddings
    retriever -.-> vector_stores
    retriever -.-> LLM
    retriever -.-> parser
    retriever -.-> all_interchangeable["all interchangeable"]
    retriever -.-> framework_glue["(framework glue + tracing)"]
```

The diagram illustrates a Retrieval-Augmented Generation (RAG) pipeline. It starts with 'your data' which is processed through 'loaders (S3, Confluence, PDFs, SQL)', 'splitters', 'embeddings', and 'vector stores'. A 'retriever' then pulls data from these sources and feeds it into a 'prompt', which is then processed by an 'LLM' and finally a 'parser'. A bracket below the retriever, LLM, and parser indicates that these components are 'all interchangeable' and part of the '(framework glue + tracing)'.

Trade-offs: frameworks speed up initial development and make components interchangeable, at the cost of additional abstraction, frequent API changes, and harder debugging. Many teams prototype with a framework and retain it only where it clearly saves effort.



On this page

Basic - [LangChain vs LlamaIndex vs Raw SDK](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-vs-llamaindex-vs-raw-sdk) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-vs-llamaindex-vs-raw-sdk) - [LangChain Setup](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-setup) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-setup) - [LangChain Basics](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-basics) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-basics)

Intermediate - [LangChain RAG Pipeline](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-rag-pipeline) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-rag-pipeline) - [LangChain Agents](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-agents) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-agents) - [LlamaIndex Setup](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#llamaindex-setup) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#llamaindex-setup) - [LlamaIndex RAG Pipeline](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#llamaindex-rag-pipeline) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#llamaindex-rag-pipeline)

Advanced - [LangChain Expression Language \(LCEL\)](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-expression-language-lcel) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langchain-expression-language-lcel) - [Custom Retrievers](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#custom-retrievers) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#custom-retrievers) - [LangSmith \(Tracing & Evals\)](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langsmith-tracing--evals) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#langsmith-tracing--evals) - [When to Use a Framework vs Raw SDK](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#when-to-use-a-framework-vs-raw-sdk) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#when-to-use-a-framework-vs-raw-sdk)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/#further-reading)

LangChain vs LlamaIndex vs Raw SDK

	LangChain	LlamaIndex	Raw SDK
Best for	Chains, agents, flexible pipelines	RAG, document Q&A, knowledge bases	Full control, production, simple use cases

	LangChain	LlamaIndex	Raw SDK
RAG	Good	Excellent	Manual but transparent
Agents	Excellent	Good	Manual via tool use
Learning curve	Steep	Medium	Low
Abstraction level	High	High	Low
Vendor lock-in	Some	Some	None
When to choose	Complex agent workflows	Document Q&A, data indexing	Most production use cases

Recommendation: Start with the raw SDK. Add a framework only when you're writing the same boilerplate repeatedly.

LangChain Setup

```

pip install langchain langchain-anthropic langchain-openai
langchain-text-splitters # LangChain 1.x
pip install faiss-cpu # local vector store
pip install langchain-chroma # Chroma vector
store

```

```

# LangChain works with both providers
from langchain_anthropic import ChatAnthropic
from langchain_openai import ChatOpenAI, OpenAIEmbeddings

llm = ChatAnthropic(model="claude-sonnet-5-5")
llm_openai = ChatOpenAI(model="gpt-6-sol")
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

```

LangChain Basics

Invoke a model

```
from langchain_anthropic import ChatAnthropic
from langchain_core.messages import HumanMessage,
SystemMessage

llm = ChatAnthropic(model="claude-sonnet-5-5")

# Simple call. Use .text for the answer: on models with
# thinking, .content is a list of blocks
# (thinking and text), while .text joins just the text
response = llm.invoke("What is Apache Kafka?")
print(response.text)

# With system message
messages = [
    SystemMessage(content="You are a concise data
engineering expert."),
    HumanMessage(content="What is Apache Kafka?"),
]
response = llm.invoke(messages)
print(response.text)
```

Prompt templates

```
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert in {domain}. Be
concise."),
    ("human", "{question}"),
])

chain = prompt | llm
response = chain.invoke({"domain": "data engineering",
"question": "What is a DAG?"})
print(response.text)
```

Output parsers

```
from langchain_core.output_parsers import StrOutputParser
from pydantic import BaseModel
langchain_core.pydantic_v1 was removed in langchain-core 0.3

# String output
chain = prompt | llm | StrOutputParser()
result = chain.invoke({"domain": "SQL", "question":
"Explain a CTE"})
print(result) # plain string

# Structured JSON output
class PipelineInfo(BaseModel):
    name: str
    schedule: str
    source: str
    destination: str

# with_structured_output uses the provider's native
structured output / tool calling –
# more reliable than asking for JSON in the prompt and
parsing it
structured_llm = llm.with_structured_output(PipelineInfo)

structured_prompt = ChatPromptTemplate.from_messages([
    ("system", "Extract the pipeline details."),
    ("human", "{text}"),
])

chain = structured_prompt | structured_llm
result = chain.invoke({"text": "Nightly job that loads
payments from the billing API into the warehouse at 2am
UTC"})
print(result) # PipelineInfo(name=..., schedule=..., ...)
```

LangChain RAG Pipeline

```

from pathlib import Path

from langchain_core.documents import Document
from langchain_text_splitters import
RecursiveCharacterTextSplitter
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

# — 1. Load documents


---


# A few lines of pathlib replace the langchain-community
loaders, a package that is being sunset
docs = [Document(page_content=path.read_text(encoding="utf-
8"), metadata={"source": str(path)})
          for path in sorted(Path("./docs").rglob("*.md"))]
print(f"Loaded {len(docs)} documents")

# — 2. Split into chunks


---


splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
    separators=["\n\n", "\n", " ", ""]
)
chunks = splitter.split_documents(docs)
print(f"Split into {len(chunks)} chunks")

# — 3. Embed and store


---


embeddings = OpenAIEmbeddings(model="text-embedding-3-
small")
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory="./chroma_db"
)

# — 4. Create retriever


---



```

```

retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 5}
)

# — 5. Build RAG chain


---



llm = ChatOpenAI(model="gpt-6-luna")

prompt = ChatPromptTemplate.from_template("""
Answer based only on the context below.
If the answer isn't there, say "I don't have that
information."

Context:
{context}

Question: {question}
""")

def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

rag_chain = (
    {"context": retriever | format_docs, "question":
    RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

# — 6. Ask questions


---



answer = rag_chain.invoke("What is the medallion
architecture?")
print(answer)

# Stream
for chunk in rag_chain.stream("Explain PySpark window
functions"):
    print(chunk, end="", flush=True)

```

With metadata filtering

```
from langchain_chroma import Chroma

retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={
        "k": 5,
        "filter": {"source": "docs/pyspark-reference.md"}
    }
)
# Chroma metadata filter
```

LangChain Agents

```
from langchain_anthropic import ChatAnthropic
from langchain.agents import create_agent  #
LangChain 1.0+
from langchain_core.tools import tool

# — Define tools with @tool decorator

@tool
def run_sql(query: str) -> str:
    """Execute a SQL SELECT query against the data
    warehouse. Returns results as JSON."""
    # Simulated
    return '{"rows": [{"count": 152340}]}'

@tool
def get_table_schema(table_name: str) -> str:
    """Get the schema (columns and types) for a given
    table."""
    schemas = {
        "orders": "order_id VARCHAR, customer_id INT,
        amount DECIMAL, status VARCHAR"
    }
    return schemas.get(table_name, f"Table {table_name} not
    found")
```

```

@tool
def check_pipeline_status(pipeline_name: str) -> str:
    """Check whether a data pipeline ran successfully
    recently."""
    return f"{pipeline_name}: last run 2024-03-15 02:15
    UTC, status=SUCCESS, rows=15234"

tools = [run_sql, get_table_schema, check_pipeline_status]

# — Build agent


---


# LangChain 1.0 replaced AgentExecutor /
create_tool_calling_agent (now in the
# langchain-classic package) with create_agent, which runs
on LangGraph.
llm = ChatAnthropic(model="claude-sonnet-5-5")

agent = create_agent(
    model=llm,
    tools=tools,
    system_prompt="You are a data engineering assistant.
    Use tools to answer accurately.",
)

result = agent.invoke(
    {"messages": [{"role": "user", "content": "How many
    orders do we have and is the pipeline healthy?"}]},
    config={"recursion_limit": 20},      # cap the number of
    agent steps
)
print(result["messages"][-1].text)      # final answer text

```

LlamaIndex Setup

```

pip install llama-index llama-index-llms-anthropic llama-
index-llms-openai

```

```
pip install llama-index-embeddings-openai llama-index-  
vector-stores-chroma
```

```
from llama_index.llms.anthropic import Anthropic  
from llama_index.llms.openai import OpenAI  
from llama_index.embeddings.openai import OpenAIEmbedding  
from llama_index.core import Settings  
  
# Global settings  
Settings.llm = Anthropic(model="claude-sonnet-5") #  
see the note below on model names  
Settings.embed_model = OpenAIEmbedding(model="text-  
embedding-3-small")  
Settings.chunk_size = 512  
Settings.chunk_overlap = 50
```

The LlamaIndex Anthropic integration checks the model name against a list built into the package, so a model released after your installed version is rejected with `ValueError: Unknown model. claude-sonnet-5-5` was rejected this way on the day it was released. Upgrade `llama-index-llms-anthropic`, or use a model the installed version knows.

LlamaIndex RAG Pipeline

```
from llama_index.core import VectorStoreIndex,  
SimpleDirectoryReader  
from llama_index.core import StorageContext,  
load_index_from_storage  
from llama_index.core.node_parser import SentenceSplitter  
from pathlib import Path  
  
# — Build index from docs directory  
_____  
documents = SimpleDirectoryReader("./docs/").load_data()  
  
parser = SentenceSplitter(chunk_size=512, chunk_overlap=50)  
nodes = parser.get_nodes_from_documents(documents)  
  
index = VectorStoreIndex(nodes)  
  
# — Save & load
```

```

index.storage_context.persist("./storage")

# Load from disk
storage_context =
StorageContext.from_defaults(persist_dir="./storage")
index = load_index_from_storage(storage_context)

# — Query

```

```

query_engine = index.as_query_engine(
    similarity_top_k=5,
    response_mode="compact"    # "compact" | "refine" |
"tree_summarize"
)

response = query_engine.query("What is the medallion
architecture?")
print(response.response)

# See source nodes
for node in response.source_nodes:
    print(f"Score: {node.score:.3f}    Source:
{node.metadata.get('file_name', 'unknown')}")
    print(f"    {node.text[:200]}\n")

# — Chat mode (multi-turn)

```

```

chat_engine =
index.as_chat_engine(chat_mode="condense_plus_context")
response = chat_engine.chat("What is Kafka?")
print(response.response)
response = chat_engine.chat("How does it compare to
RabbitMQ?")
print(response.response)

```

With metadata filtering



```

from llama_index.core.vector_stores import MetadataFilter,
MetadataFilters

filters = MetadataFilters(filters=[
    MetadataFilter(key="file_name", value="pyspark-
reference.md")
])
query_engine = index.as_query_engine(filters=filters,
similarity_top_k=3)
response = query_engine.query("Explain window functions")

```

Sub-question query engine (multi-document)

```

from llama_index.core import Settings
from llama_index.core.query_engine import
SubQuestionQueryEngine
from llama_index.core.question_gen import
LLMQuestionGenerator
from llama_index.core.tools import QueryEngineTool

# Build separate indexes per document
pyspark_engine =
VectorStoreIndex.from_documents(pyspark_docs).as_query_engi
ne()
airflow_engine =
VectorStoreIndex.from_documents(airflow_docs).as_query_engi
ne()

tools = [

QueryEngineTool.from_defaults(query_engine=pyspark_engine,
name="pyspark",
description="PySpark
documentation"),

QueryEngineTool.from_defaults(query_engine=airflow_engine,
name="airflow",
description="Airflow
documentation"),
]

```

```
# Pass the question generator explicitly. The default one
needs the llama-index-question-gen-openai package,
# which pins an older llama-index-core and breaks the
Anthropic integration when installed next to it
sub_question_engine = SubQuestionQueryEngine.from_defaults(
    query_engine_tools=tools,

    question_gen=LLMQuestionGenerator.from_defaults(llm=Setting
s.llm),
)
response = sub_question_engine.query(
    "Compare Spark Structured Streaming with Airflow
scheduling for batch workloads"
)
print(response.response)
```

LangChain Expression Language (LCEL)

LCEL uses the `|` operator to compose chains — similar to Unix pipes.

```
from langchain_core.runnables import RunnableParallel,
RunnableLambda

# — Sequential chain
—
chain = prompt | llm | StrOutputParser()

# — Parallel execution
—
parallel_chain = RunnableParallel({
    "summary": prompt_summary | llm |
StrOutputParser(),
    "key_points": prompt_bullets | llm |
StrOutputParser(),
})
result = parallel_chain.invoke({"text": long_document})

# — With fallback
—
```

```

primary = ChatAnthropic(model="claude-sonnet-5-5")
fallback = ChatOpenAI(model="gpt-6-luna")

chain_with_fallback = (prompt | primary |
StrOutputParser()).with_fallbacks(
    [prompt | fallback | StrOutputParser()]
)

# — Add retry logic

```

```

reliable_llm = llm.with_retry(stop_after_attempt=3,
wait_exponential_jitter=True)

# — Conditional routing

```

```

def route(inputs):
    if "sql" in inputs["question"].lower():
        return sql_chain
    elif "pipeline" in inputs["question"].lower():
        return pipeline_chain
    return general_chain

routed_chain = RunnableLambda(route)

```

Custom Retrievers

```

from langchain_core.retrievers import BaseRetriever
from langchain_core.documents import Document

class HybridRetriever(BaseRetriever):
    """Combines vector search and keyword search."""

    vectorstore: object
    bm25_index: object
    k: int = 5
    alpha: float = 0.5 # 0=BM25, 1=vector

    def _get_relevant_documents(self, query: str) ->

```

```

list[Document]:
    # Vector results
    vec_docs =
self.vectorstore.similarity_search(query, k=self.k * 2)
    # BM25 results
    bm25_docs =
self.bm25_index.get_top_n(query.split(), self.k * 2)

    # Reciprocal Rank Fusion
    scores = {}
    for rank, doc in enumerate(vec_docs):
        key = doc.page_content[:50]
        scores[key] = scores.get(key, 0) + self.alpha *
(1 / (rank + 1))
    for rank, doc in enumerate(bm25_docs):
        key = doc.page_content[:50]
        scores[key] = scores.get(key, 0) + (1 -
self.alpha) * (1 / (rank + 1))

    all_docs = {d.page_content[:50]: d for d in
vec_docs + bm25_docs}
    ranked = sorted(scores.items(), key=lambda x:
x[1], reverse=True)
    return [all_docs[k] for k, _ in ranked[:self.k] if
k in all_docs]

```

LangSmith (Tracing & Evals)

LangSmith traces every LLM call — inputs, outputs, latency, cost — for debugging and evaluation.

```
pip install langsmith openevals
```

```

import os
os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_API_KEY"] = "lsv2_..." #
from the LangSmith settings page
os.environ["LANGSMITH_PROJECT"] = "de-handbook-rag" # the
older LANGCHAIN_* names are still read

```

```
# All LangChain calls are now automatically traced
result = rag_chain.invoke("What is Kafka?")
# Visit https://smith.langchain.com to see the trace
```

```
# Run evaluations
from langsmith import Client, evaluate
from openevals.llm import create_llm_as_judge
from openevals.prompts import CORRECTNESS_PROMPT

client = Client()

# Create a dataset of questions and reference answers
dataset = client.create_dataset(dataset_name="de-rag-eval")
client.create_examples(
    dataset_id=dataset.id,
    examples=[
        {"inputs": {"question": "What is the medallion architecture?"},
         "outputs": {"answer": "Bronze/Silver/Gold layers"}},
        {"inputs": {"question": "How does Kafka guarantee delivery?"},
         "outputs": {"answer": "At-least-once by default, exactly-once with transactions"}}],
)

# An LLM-as-judge evaluator. The model string is "provider:model-name"
judge = create_llm_as_judge(prompt=CORRECTNESS_PROMPT,
                             feedback_key="correctness",
                             model="anthropic:claude-sonnet-5-5")

def correctness(inputs: dict, outputs: dict,
                reference_outputs: dict):
    return judge(inputs=inputs, outputs=outputs,
                 reference_outputs=reference_outputs)

# Evaluate: run the chain on every example and score each answer
results = evaluate(
```

```
lambda inputs: {"answer":  
rag_chain.invoke(inputs["question"])},  
    data=dataset.name,  
    evaluators=[correctness],  
    experiment_prefix="rag-v1",  
)
```

An evaluator can also be a plain function that returns a score, for example a check that the answer cites a source. `LangChainStringEvaluator`, which older versions of this guide used, is no longer part of `langsmith`.

When to Use a Framework vs Raw SDK

Use the raw SDK when:

- Simple use case (single LLM call, basic RAG)
- Production code where you need full control
- You're optimizing for latency or cost
- Your team doesn't already know the framework

Use LangChain when:

- Building complex agent workflows
- Need to swap LLM providers easily
- Want built-in streaming, retries, fallbacks
- Already using LangSmith for tracing

Use LlamaIndex when:

- Building document Q&A or knowledge base
- Need advanced retrieval (sub-question, HyDE, re-ranking)
- Working with large document collections
- Want higher-level RAG abstractions out of the box

Common Pitfalls

Pitfall	Symptom	Fix
Copying tutorials written for older versions	<code>ImportError</code> s: <code>langchain.text_splitter</code> , <code>pydantic_v1</code> , <code>ServiceContext</code> , <code>AgentExecutor</code>	Check the version; use the split packages (<code>langchain-*</code> integrations, <code>langchain-text-splitters</code>) and current APIs (<code>create_agent</code> , <code>Settings</code>)
Unpinned framework versions	A minor upgrade breaks production	Pin exact versions; upgrade deliberately with tests
Default chunking and retrieval settings	Mediocre answers that no prompt fixes	Tune chunk size, k, and re-ranking on an eval set
Parsing JSON from free text	Intermittent parse failures	<code>llm.with_structured_output(PydanticModel)</code> (LangChain) or structured outputs in LlamaIndex
Abstractions hiding the actual prompt	Can't tell why the model behaved oddly	Turn on tracing (LangSmith, Langfuse, OpenTelemetry) and read the real prompts
Agents with no step limit	Runaway loops and bills	<code>recursion_limit</code> / <code>max_iterations</code> , and cost guards
Rebuilding the index on every app start	Slow startup; repeated embedding costs	Persist the index or vector store; load it instead of rebuilding
Using a framework for a single API call	Extra dependencies and complexity for no gain	Call the provider SDK directly

Cheat Sheet

LangChain (1.x)

Task	Code
Chat model	<code>ChatAnthropic(model="claude-sonnet-5")</code> · <code>ChatOpenAI(model=...)</code>

Task	Code
Call it	<code>llm.invoke("question") → .text · llm.stream(...) · llm.batch([...])</code>
Prompt + model + parser (LCEL)	<code>chain = prompt llm StrOutputParser() → chain.invoke({...})</code>
Structured output	<code>llm.with_structured_output(MyPydanticModel)</code>
Split text	<code>RecursiveCharacterTextSplitter(chunk_size=800, chunk_overlap=100).split_documents(docs)</code>
Vector store → retriever	<code>Chroma.from_documents(docs, embeddings).as_retriever(search_kwargs={"k": 5})</code>
Tool	<code>@tool</code> on a typed function with a docstring
Agent	<code>create_agent(model=llm, tools=[...], system_prompt="...") → .invoke({"messages": [...]})</code>
Fallback model	<code>primary.with_fallbacks([backup])</code>
Retries	<code>llm.with_retry(stop_after_attempt=3)</code>

LlamaIndex

Task	Code
Global model settings	<code>Settings.llm = Anthropic(model="claude-sonnet-5") · Settings.embed_model = ...</code>
Load documents	<code>SimpleDirectoryReader("docs/").load_data()</code>
Build an index	<code>VectorStoreIndex.from_documents(docs)</code>
Ask questions	<code>index.as_query_engine(similarity_top_k=5).query("...")</code>
Chat over data	<code>index.as_chat_engine()</code>
Persist / load	<code>index.storage_context.persist("./storage") · load_index_from_storage(StorageContext.from_defaults(persist_dir="./storage"))</code>

Task	Code
Filter	<pre>MetadataFilters(filters=[MetadataFilter(key="source", value="runbooks")])</pre>

Choosing: a single call or simple extraction → provider SDK · RAG over many documents → LlamaIndex · multi-step agents with state, branching, and human approval → LangGraph / LangChain · heavy production requirements → often your own thin layer on the SDK

Interview Questions

Q: When would you use LangChain or LlamaIndex, and when would you use the raw SDK?

A: Use a framework when it removes real work: many document loaders and vector-store integrations, standard RAG patterns, provider swapping, or agent orchestration with state and checkpoints. Use the raw SDK for simple or performance-critical paths where you want full control over prompts, retries, and costs, fewer dependencies, and the newest provider features the day they launch. A common pattern is to prototype with a framework and harden the critical path on the SDK.

Q: What is LCEL? A: The LangChain Expression Language composes components ("runnables") with the pipe operator — `prompt | llm | parser` — into a chain. Every runnable has the same interface (`invoke`, `batch`, `stream`, and `async` versions), so a composed chain automatically supports streaming, batching, parallel branches (`RunnableParallel`), fallbacks, and retries, and shows up in tracing.

Q: How does LlamaIndex build and query an index? A: It loads documents into `Document` objects, splits them into nodes (chunks) with a node parser, embeds each node, and stores them in a vector store behind a `VectorStoreIndex`. At query time, a retriever finds the top-k similar nodes (optionally filtered and re-ranked), and a response synthesizer puts them into a prompt and calls the LLM. Query engines like the sub-question engine break complex questions into smaller ones across several indexes.

Q: How do you debug a LangChain or LlamaIndex application? A: Tracing first: LangSmith, Langfuse, or an OpenTelemetry-based tool shows every step — the rendered prompts, retrieved chunks, tool calls, token counts, and latencies. Check retrieval separately from generation (did the right chunks come back?), reproduce failures with the exact inputs from the trace, and add them to an evaluation set so the fix is locked in.

Further Reading

- [LangChain documentation](https://docs.langchain.com/) (https://docs.langchain.com/) and [LangGraph](https://langchain-ai.github.io/langgraph/) (https://langchain-ai.github.io/langgraph/)
- [LangChain v1 migration guide](https://docs.langchain.com/oss/python/migrate/langchain-v1) (https://docs.langchain.com/oss/python/migrate/langchain-v1)
- [LlamaIndex documentation](https://developers.llamaindex.ai/python/framework/) (https://developers.llamaindex.ai/python/framework/)
- [LangSmith](https://docs.langchain.com/langsmith/observability) (https://docs.langchain.com/langsmith/observability) — tracing and evaluation
- [Anthropic: Building effective agents](https://www.anthropic.com/engineering/building-effective-agents) (https://www.anthropic.com/engineering/building-effective-agents) — when frameworks help and when they hide too much

Previous: [MCP and Text-to-SQL](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mcp-text-to-sql/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mcp-text-to-sql/) • **Next:** [Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026