```
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

urls = [
    "https://lilianweng.github.io/posts/2023-06-23-agent/",
    "https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/",
    "https://lilianweng.github.io/posts/2023-10-25-adv-attack-llm/",
]

docs = [WebBaseLoader(url).load() for url in urls]
docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)
doc_splits = text_splitter.split_documents(docs_list)

# Add to vectorDB
vectorstore = Chroma.from_documents(
    documents=doc_splits,
    collection_name="rag-chroma",
    embedding=OpenAIEmbeddings(),
)
retriever = vectorstore.as_retriever()
```

Code Explanation:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

Imports a text chunker

Its job: split long documents into smaller chunks

“Recursive” means:

It tries paragraph → sentence → word boundaries

Avoids cutting text in unnatural places

```
from langchain_community.document_loaders import WebBaseLoader
```

Imports a **web page loader**

It:

- Fetches a URL
- Extracts readable text (HTML → text)
- Wraps it in LangChain `Document` objects

```
from langchain_community.vectorstores import Chroma
```

Imports **Chroma**, a vector database

Used to:

- Store embeddings
- Perform similarity search later

```
from langchain_openai import OpenAIEmbeddings
```

Imports the embedding model wrapper

This converts text → vectors using OpenAI embeddings

Required before storing anything in a vector DB

```
urls = [
    "https://lilianweng.github.io/posts/2023-06-23-agent/",
    "https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/",
    "https://lilianweng.github.io/posts/2023-10-25-adv-attack-llm/",]
```

A list of **blog post URLs**

These are your **knowledge sources**
Each page will become searchable later

```
docs = [WebBaseLoader(url).load() for url in urls]
```

Load the web pages:

What happens for **each URL**:

1. `WebBaseLoader(url)` → creates a loader
2. `.load()`:
    - Downloads the page
    - Extracts text
    - Returns a **list of `Document` objects**

Important:

- `.load()` returns a **list**, not a single document
- So `docs` becomes:

```
[
  [Document(...), Document(...)],  # page 1
  [Document(...)],                 # page 2
  [Document(...)]                  # page 3
]
```

That's a **list of lists**.

Flatten the list

```
docs_list = [item for sublist in docs for item in sublist]
```

This:

- Flattens the nested list

- Converts:

```
[[doc1, doc2], [doc3], [doc4]]
```

Into:

```
[doc1, doc2, doc3, doc4]
```

Why this matters:

- Text splitters expect a **flat list of `Document` objects**

Create the text splitter

```
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)
```

This configures **how documents are chunked**.

Details:

- `from_tiktoken_encoder`:
  - Uses OpenAI's tokenizer (`tiktoken`)
  - Chunk size is measured in **tokens**, not characters

- `chunk_size=250`:
  - Each chunk ≈ 250 tokens

- `chunk_overlap=0`:
  - No overlap between chunks
  - Simpler, but less context sharing

Result:

- You get **token-aware chunks**, which is good for LLMs

Split documents into chunks

```
doc_splits = text_splitter.split_documents(docs_list)
```

What this does:

- Takes each `Document`
- Breaks its text into multiple smaller `Document` chunks
- Preserves metadata (like source URL)

Before:

```
len(docs_list) = maybe 5 documents
```

After:

```
len(doc_splits) = maybe 100+ chunks
```

Each chunk:

- Has `.page_content`
- Has `.metadata` (source, etc.)

Create the vector store

```
vectorstore = Chroma.from_documents(
    documents=doc_splits,
    collection_name="rag-chroma",
    embedding=OpenAIEmbeddings(),
)
```

This is the **core RAG step**.

What happens internally:

1. Each chunk's text is sent to **OpenAIEmbeddings**
2. Each chunk becomes a **vector**
3. Vectors are stored in **Chroma**
4. They are grouped under:

```
collection_name = "rag-chroma"
```

After this:

- Your blog posts are now **searchable by meaning**, not keywords

Create a retriever

```
retriever = vectorstore.as_retriever()
```

This converts the vector store into a **retrieval interface**.

Meaning:

- You can now do:

```
retriever.invoke("agent memory")
```

And it will:

1. Embed the query
2. Compare it to stored vectors
3. Return the **most relevant chunks**

This `retriever` is what you plug into:

- RAG chains
- LangGraph nodes
- CRAG pipelines

Mental model (important)

This whole script does **one thing**:

> **Turn web pages → chunks → embeddings → searchable memory**

```python
### Retrieval Grader

from langchain_core.prompts import ChatPromptTemplate

from pydantic import BaseModel, Field

from langchain_openai import ChatOpenAI

# Data model

class GradeDocuments(BaseModel):

    """Binary score for relevance check on retrieved documents."""



    binary_score: str = Field(

        description="Documents are relevant to the question, 'yes' or 'no'"

    )




# LLM with function call

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

structured_llm_grader = llm.with_structured_output(GradeDocuments)



# Prompt

system = """You are a grader assessing relevance of a retrieved document to a user question. \n

    If the document contains keyword(s) or semantic meaning related to the question, grade it as relevant. \n

    Give a binary score 'yes' or 'no' score to indicate whether the document is relevant to the question."""
```

```
grade_prompt = ChatPromptTemplate.from_messages(

    [

        ("system", system),

        ("human", "Retrieved document: \n\n {document} \n\n User question:
{question}"),

    ]

)



retrieval_grader = grade_prompt | structured_llm_grader

question = "agent memory"

#docs = retriever.get_relevant_documents(question)

docs = retriever.invoke(question)



doc_txt = docs[1].page_content

print(retrieval_grader.invoke({"question": question, "document": doc_txt}))
```

```
from langchain_core.prompts import ChatPromptTemplate
```

Imports LangChain's prompt builder that lets you define a prompt as a sequence of messages (system/human).

```
from pydantic import BaseModel, Field
```

Imports Pydantic so you can define a **structured schema** for the model output.

```
from langchain_openai import ChatOpenAI
```

Imports the OpenAI chat model wrapper you're using.

Define the structured output schema

```
class GradeDocuments(BaseModel):
    """Binary score for relevance check on retrieved documents."""
```

Defines a Pydantic model. This tells LangChain: "the LLM must output JSON that matches this structure."

```
binary_score: str = Field(
    description="Documents are relevant to the question, 'yes' or 'no'"
)
```

Your schema has one field, `binary_score`, which must be a string like `"yes"` or `"no"`.

Build an LLM that outputs that schema

```
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
```

Creates a chat model instance. `temperature=0` makes it deterministic-ish.

```
structured_llm_grader = llm.with_structured_output(GradeDocuments)
```

Wraps the LLM so that when you call it, it **must** return output parsable into `GradeDocuments`.

Under the hood, LangChain will:

- instruct the model to return structured JSON
- parse it into a `GradeDocuments` object

Build the grading prompt

```
system = """You are a grader assessing relevance of a retrieved document to a
user question. \n
...
"""
```

This is the system instruction telling the model how to behave: judge relevance and output yes/no.

```
grade_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system),
        ("human", "Retrieved document: \n\n {document} \n\n User
question: {question}"),
    ]
)
```

Creates a prompt template with:

- a **system message** (the rules)
- a **human message** that injects two variables:
    - `{document}`: the retrieved chunk text
    - `{question}`: the user question

Compose into a runnable chain

```
retrieval_grader = grade_prompt | structured_llm_grader
```

This uses LangChain Expression Language (LCEL):

- first apply `grade_prompt` to format messages
- then send to `structured_llm_grader`
  Result: `retrieval_grader.invoke({...})` returns a `GradeDocuments` instance.

Retrieve documents

```
question = "agent memory"
```

The query you'll use for retrieval and grading.

```
docs = retriever.invoke(question)
```

Runs vector search:

- embeds the question
- finds nearest chunks
- returns a list of `Document` objects

Pick one chunk and grade it

```
doc_txt = docs[1].page_content
```

Takes the **second** retrieved document and extracts its text.

⚠️ This can crash if fewer than 2 docs are returned.

```
print(retrieval_grader.invoke({"question": question, "document": doc_txt}))
```

Calls the chain:

- fills prompt variables
- model returns structured output
- LangChain parses it into `GradeDocuments`

You'll print something like:

```
GradeDocuments(binary_score='yes')
```

Fix 1: avoid index error

Replace:

```
doc_txt = docs[1].page_content
```

with:

```
doc_txt = docs[0].page_content if docs else ""
```

```
### Generate

from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Prompt - RAG prompt template (equivalent to rlm/rag-prompt from hub)
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an assistant for question-answering tasks. Use the
following pieces of retrieved context to answer the question. If you don't know
the answer, just say that you don't know. Use three sentences maximum and keep
the answer concise."),
    ("human", "Question: {question}\n\nContext: {context}")
])

# LLM
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)


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


# Chain - includes retrieval, formatting, prompt, LLM, and output parsing
from langchain_core.runnables import RunnablePassthrough, RunnableLambda

rag_chain = (
    {
        "context": RunnableLambda(lambda x: x["question"]) | retriever |
format_docs,
        "question": RunnableLambda(lambda x: x["question"])
    }
    | prompt
    | llm
    | StrOutputParser()
)

# Run
generation = rag_chain.invoke({"question": question})
```

```
print(generation)
```

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

`ChatPromptTemplate`: builds a prompt as chat messages (system/human) with variables like `{question}` and `{context}`.
`StrOutputParser`: converts the LLM output into a plain Python `str` (instead of a message object)

2) Prompt template (the "instructions + slots")

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an assistant for question-answering tasks. Use
the following pieces of retrieved context to answer the question. If you
don't know the answer, just say that you don't know. Use three sentences
maximum and keep the answer concise."),
    ("human", "Question: {question}\n\nContext: {context}")
])
```

This creates a prompt with **two messages**:

- **System message**: sets behavior rules (use context, max 3 sentences, say "I don't know" if needed).
- **Human message**: contains placeholders:
    - `{question}` will be replaced by the user question
    - `{context}` will be replaced by retrieved text chunks

So later, the chain must produce a dict with keys:

```
{"question": ..., "context": ...}
```

3) LLM

```
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)
```

- Creates the chat model to generate the final answer.
- `temperature=0` → more deterministic, less creative.

(Important: this assumes you already imported `ChatOpenAI` somewhere: `from langchain_openai import ChatOpenAI`.)

4) Post-processing: format retrieved documents into a string

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

- Input: `docs` = list of `Document` objects returned by `retriever`
- Each `Document` has:
  - `.page_content` (the chunk text)
  - `.metadata` (source, etc.)
- Output: one big string containing all chunk texts separated by blank lines.

Example output shape:

```
chunk1 text...

chunk2 text...

chunk3 text...
```

This becomes `{context}` in the prompt.

5) Runnable imports

```
from langchain_core.runnables import RunnablePassthrough, RunnableLambda
```

`RunnableLambda`: wraps a Python function so it can be used inside LangChain's runnable pipeline (`|` operator).

`RunnablePassthrough`: not used in your code; it just passes input through unchanged.

6) The RAG chain (the real pipeline)

```
rag_chain = (
    {
        "context": RunnableLambda(lambda x: x["question"]) | retriever |
format_docs,
        "question": RunnableLambda(lambda x: x["question"])
    }
    | prompt
    | llm
    | StrOutputParser()
)
```

This is a pipeline made of **stages** connected by `|`.

6.1 The first stage is a "mapping" step (build inputs for the prompt)

You create a dict with two keys: `"context"` and `"question"`.

Key: `"question"`

```
"question": RunnableLambda(lambda x: x["question"])
```

- Input to the chain: `x` is a dict, like `{"question": "agent memory"}`
- This extracts the value: `"agent memory"`
- Output for this key: the actual question string

Key: `"context"`

```
"context": RunnableLambda(lambda x: x["question"]) | retriever |
format_docs
```

This is a 3-step mini-pipeline:

1. `RunnableLambda(lambda x: x["question"])`
    - Extracts the question string from the input dict.

2. `| retriever`
    - Takes that question string
    - Runs vector search in your vector DB
    - Output: list of `Document` chunks (top-k results)
3. `| format_docs`
    - Takes list of `Document`
    - Converts to one string by joining `.page_content`

So `"context"` becomes a big text string built from retrieved chunks.

Output after this mapping stage

At this point, the chain produces:

```
{
  "question": "agent memory",
  "context": "chunk1...\n\nchunk2...\n\nchunk3..."
}
```

7) Apply the prompt template

```
| prompt
```

- Takes the dict with `question` and `context`
- Fills `{question}` and `{context}`
- Produces chat messages ready to send to the LLM.

8) Call the LLM

```
| llm
```

- Sends the formatted messages to the model
- Output: an AI message object (model response)

---

9) Parse to string

`| StrOutputParser()`

- Converts the model output into a plain Python string (`str`).

10) Run the chain

```
generation = rag_chain.invoke({"question": question})
print(generation)
```

- Calls the whole pipeline end-to-end.
- Input is a dict with a `"question"` key.
- Output is a string answer.
- Prints it.

```python
from typing import List

from typing_extensions import TypedDict


class GraphState(TypedDict):
    """
    Represents the state of our graph.

    Attributes:
        question: question
        generation: LLM generation
        web_search: whether to add search
        documents: list of documents
    """

    question: str
    generation: str
    web_search: str
    documents: List[str]
```

```
from typing import List
```

Imports the `List` type for type hints.

Used to say “this variable is a list of X”.

```
from typing_extensions import TypedDict
```

Imports `TypedDict`, which lets you define the **shape of a dictionary**.

Think of it as a **schema for a Python dict**.

LangGraph relies on this to know what keys exist in the graph state.

Define the graph state schema

```
class GraphState(TypedDict):
```

This defines a **state object** for a LangGraph graph.

Important mental model:

> In LangGraph, **state = one dictionary that flows through all nodes**.

Each node:

- reads from the state
- writes updates back to the state

`web_search: str`

A **control flag** that guides graph routing

- Often used as:
  - `"yes"` / `"no"`
  - or `"true"` / `"false"`

Typical use:

- A grader node sets this
- A router node checks it
- If `"yes"` → run a web search node
- If `"no"` → skip it

This is what makes the graph **conditional**, not linear.

Why this matters in LangGraph

LangGraph nodes **must agree on the state structure**.

This `GraphState`:

- tells LangGraph what keys exist
- prevents silent bugs (wrong key names)
- makes routing and reducers possible

Key mental model (very important)

> **LangGraph = state machine**
>
> `GraphState` defines the memory of that machine.

Without this:

- nodes don't know what to expect
- routing logic becomes fragile
- debugging is painful

```python
from langchain_core.documents import Document
```

```
def retrieve(state):
    """
    Retrieve documents

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): New key added to state, documents, that contains retrieved
documents
    """
    print("---RETRIEVE---")
    question = state["question"]

    # Retrieval
    documents = retriever.invoke(question)
    return {"documents": documents, "question": question}



def generate(state):
    """
```

```
    Generate answer

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): New key added to state, generation, that contains LLM
generation
    """
    print("---GENERATE---")
    question = state["question"]
    documents = state["documents"]

    # RAG generation
    generation = rag_chain.invoke({"context": documents, "question": question})
    return {"documents": documents, "question": question, "generation":
generation}


def grade_documents(state):
    """
    Determines whether the retrieved documents are relevant to the question.
```

```
    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates documents key with only filtered relevant documents
    """

    print("---CHECK DOCUMENT RELEVANCE TO QUESTION---")
    question = state["question"]
    documents = state["documents"]

    # Score each doc
    filtered_docs = []
    web_search = "No"
    for d in documents:
        score = retrieval_grader.invoke(
            {"question": question, "document": d.page_content}
        )
        grade = score.binary_score
        if grade == "yes":
            print("---GRADE: DOCUMENT RELEVANT---")
            filtered_docs.append(d)
        else:
```

```
            print("---GRADE: DOCUMENT NOT RELEVANT---")
            web_search = "Yes"
            continue
    return {"documents": filtered_docs, "question": question, "web_search":
web_search}



def transform_query(state):
    """
    Transform the query to produce a better question.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates question key with a re-phrased question
    """

    print("---TRANSFORM QUERY---")
    question = state["question"]
    documents = state["documents"]

    # Re-write question
```

```
    better_question = question_rewriter.invoke({"question": question})
    return {"documents": documents, "question": better_question}



def web_search(state):
    """
    Web search based on the re-phrased question.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): Updates documents key with appended web results
    """

    print("---WEB SEARCH---")
    question = state["question"]
    documents = state["documents"]

    # Web search
    docs = web_search_tool.invoke({"query": question})
    web_results = "\n".join([d["content"] for d in docs])
    web_results = Document(page_content=web_results)
```

```
    documents.append(web_results)

    return {"documents": documents, "question": question}


### Edges


def decide_to_generate(state):
    """
    Determines whether to generate an answer, or re-generate a question.

    Args:
        state (dict): The current graph state

    Returns:
        str: Binary decision for next node to call
    """

    print("---ASSESS GRADED DOCUMENTS---")
    state["question"]
    web_search = state["web_search"]
    state["documents"]
```

```
    if web_search == "Yes":
        # All documents have been filtered check_relevance
        # We will re-generate a new query
        print(
            "---DECISION: ALL DOCUMENTS ARE NOT RELEVANT TO QUESTION, TRANSFORM QUERY---"
        )
        return "transform_query"
    else:
        # We have relevant documents, so generate answer
        print("---DECISION: GENERATE---")
        return "generate"
```

Register nodes (functions)

Each node is a **pure Python function** that:

- takes `state`
- returns a `dict` with updates

```
workflow.add_node("retrieve", retrieve)
```

Adds a node named "`retrieve`" that runs the `retrieve(state)` function.

```
workflow.add_node("grade_documents", grade_documents)
```

Adds a node that evaluates document relevance.

```
workflow.add_node("generate", generate)
```

Adds a node that produces the final answer with the LLM.

```
workflow.add_node("transform_query", transform_query)
```

Adds a node that rewrites the question if retrieval failed.

```
workflow.add_node("web_search_node", web_search)
```

Adds a node that performs live web search and appends results.

Important:

- Node names are **strings**
- They're used later when defining edges

```
workflow.add_conditional_edges(
    "grade_documents",
    decide_to_generate,
    {
        "transform_query": "transform_query",
        "generate": "generate",
    },
)
```

This is the **decision point**.

How this works:

1. After `grade_documents` runs
2. LangGraph calls:

```
decide_to_generate(state)
```

That function returns a **string**, e.g.:

- `"Transform_query"`
- `"generate"`

LangGraph looks up that string in the mapping:

```
{
    "transform_query": "transform_query",
    "generate": "generate",
}
```

Execution continues to the mapped node

Meaning in plain English:

- If documents are irrelevant → rewrite the question
- If documents are relevant → generate the answer

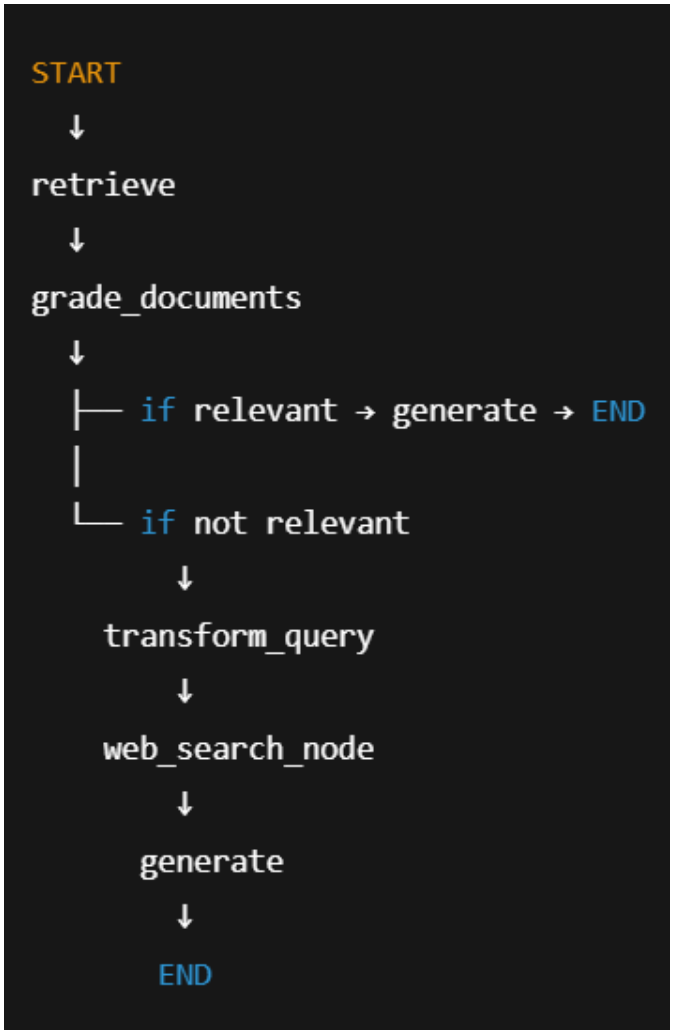
This is what makes the graph **non-linear and intelligent**.

Recovery path: rewrite → web search

If the graph went down the `"transform_query"` path:

- It **always** follows up with web search

So rewritten question → fresh external context


START
retrieve
grade_documents
if relevant → generate → END
if not relevant
transform_query
web_search_node
generate
END