```
from dotenv import load_dotenv
import os

load_dotenv()  # reads .env and injects into environment
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_ENDPOINT"] = "https://api.smith.langchain.com"

# retrieve from environment
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY")
```

Import utilities

```
from dotenv import load_dotenv
import os
```

- `dotenv.load_dotenv`
  Reads a `.env` file and loads variables into the process environment.
- `os`
  Python's interface to environment variables (`os.environ`).

This combo is the standard way to handle **secrets and config** without hard-coding them.

Load variables from .env

```
load_dotenv()  # reads .env and injects into environment
```

What this does:

- Looks for a file named `.env` (usually in your project root)
- Reads lines like:

```
OPENAI_API_KEY=sk-...
LANGCHAIN_API_KEY=lsv2_...
```

- Injects them into the runtime environment so Python can access them

After this line:

```
os.getenv("OPENAI_API_KEY")
```

will return the value from `.env`.

Important:

- `.env` should **never** be committed to GitHub
- This is how you keep secrets out of code

Enable LangChain tracing (LangSmith)

```
os.environ["LANGCHAIN_TRACING_V2"] = "true"
```

This turns on **LangChain tracing**.

What that means:

- Every LLM call
- Every chain step
- Every graph node execution

…will be logged to **LangSmith**.

If this is `"false"` or missing:

- Your code still works
- But you get **zero observability**

This is essential for debugging LangGraph flows.

Set the LangSmith endpoint

```
os.environ["LANGCHAIN_ENDPOINT"] = "https://api.smith.langchain.com"
```

This tells LangChain **where to send tracing data**.

- `api.smith.langchain.com` = LangSmith's backend
- Required for tracing to work

Without this:

- Tracing may silently fail
- Or default to a wrong endpoint

Inject LangChain API key

```
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
```

What's happening:

1. `os.getenv("LANGCHAIN_API_KEY")`
    - Reads the value loaded from `.env`
2. `os.environ["LANGCHAIN_API_KEY"] = ...`
    - Explicitly sets it in the environment

Why this looks redundant but isn't:

- Some libraries check `os.environ` directly
- This guarantees the key is present at runtime
- Useful in notebooks / subprocesses / reloads

If this key is missing:

- LangChain still runs
- But LangSmith tracing **will not work**

Inject OpenAI API key

```
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY")
```

Same pattern as above, but for OpenAI.

This is required for:

- `ChatOpenAI`
- `OpenAIEmbeddings`
- Any OpenAI-backed model

Without it:

- First model call will fail with an auth error

```python
import bs4
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI, OpenAIEmbeddings


#### INDEXING ####


# Load Documents
loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("post-content", "post-title", "post-header")
        )
    ),
)
docs = loader.load()


# Split
```

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=200)

splits = text_splitter.split_documents(docs)



# Embed

vectorstore = Chroma.from_documents(documents=splits,

                                    embedding=OpenAIEmbeddings())



retriever = vectorstore.as_retriever()

#### RETRIEVAL and GENERATION ####



# Prompt - RAG prompt template (equivalent to rlm/rag-prompt from hub)

from langchain_core.prompts import ChatPromptTemplate



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
from langchain_core.runnables import RunnableLambda

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

# Question
rag_chain.invoke({"question": "What is Task Decomposition?"})
```

Imports (what tools you're bringing in)

```
import bs4
```

Imports **BeautifulSoup**

Used for **HTML parsing**

You'll use it to extract only specific parts of a webpage (not menus, ads, etc.)

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

Text chunker

Splits long text into **LLM-friendly chunks**

"Recursive" = tries paragraph → sentence → word boundaries

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

Loads content from web pages

Converts HTML → text → LangChain `Document` objects

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

Vector database

Stores embeddings and enables similarity search

```
from langchain_core.output_parsers import StrOutputParser
```

Converts LLM output into a plain Python `str`

```
from langchain_core.runnables import RunnablePassthrough
```

(Not used later — safe to remove)

Passes inputs through unchanged in chains

```
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
```

ChatOpenAI → LLM for generation

OpenAIEmbeddings → converts text into vectors

INDEXING PHASE (build searchable memory)

Load the web document

```
loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
```

Defines which URL(s) to load
Tuple allows multiple URLs

```
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("post-content", "post-title", "post-header")
        )
    ),
)
```

This is **important**.

- Tells BeautifulSoup:
  - “Only extract HTML elements with these classes”
- Prevents loading:
  - navigation bars
  - Footers
  - Ads
  - unrelated text

Result: **clean, relevant article content**

```
docs = loader.load()
```

What happens:

- Downloads the page
- Extracts selected HTML
- Returns:

```
List[Document]
```

Each `Document` contains:

- `page_content` → text
- `metadata` → source URL, etc.

Split the document into chunks

```
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200
)
```

Each chunk ≈ 1000 characters

Overlap of 200 characters:

- preserves context across chunks
- reduces “cutoff” issues

```
splits = text_splitter.split_documents(docs)
```

Transforms:

```
1 long document
```

Into:

```
many smaller Document chunks
```

Each chunk is still a `Document`.

Embed and store in vector DB

```
vectorstore = Chroma.from_documents(
    documents=splits,
    embedding=OpenAIEmbeddings()
)
```

What happens internally:

1. Each chunk → embedding vector
2. Vectors stored in Chroma
3. Metadata preserved

Now your article is **searchable by meaning**.

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

Creates a **retriever interface**:

- Input: text query
- Output: top-k relevant Document chunks

This is what RAG uses.

RETRIEVAL + GENERATION PHASE (runtime)

Prompt definition

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

Builds chat-style prompts with variables

```
prompt = ChatPromptTemplate.from_messages([
```

Defines a **two-message prompt**.

```
("system", "You are an assistant for question-answering tasks...")
```

System message:

- Defines behavior
- Limits length
- Forces honesty if context is missing

```
("human", "Question: {question}\n\nContext: {context}")
```

Human message with placeholders:

- {question} → user query
- {context} → retrieved chunks

LLM setup

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

Deterministic answers

No creativity needed for factual RAG

Post-processing retrieved docs

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

Input:

```
List[Document]
```

Output:

```
single string of chunk texts
```

This becomes {context} in the prompt.

The RAG chain (data flow)

```
from langchain_core.runnables import RunnableLambda
```

Wraps Python functions into chainable components

```
rag_chain = (
```

Defines a pipeline using **LangChain Expression Language (LCEL).**

Build inputs for the prompt

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

Input to chain:

```
{"question": "..."}
```

"question"

- Extracts question string directly

"context"

1. Extract question
2. Retrieve relevant chunks
3. Format chunks into text

Produces:

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

Prompt → LLM → Output parsing

```
| prompt
| llm
| StrOutputParser()
```

Prompt fills {question} and {context}

LLM generates answer

Parser returns a plain string

Run the chain

```
rag_chain.invoke({"question": "What is Task Decomposition?"})
```

Big mental model

This code implements a **classic RAG loop**:

```
Web page
 → clean extraction
 → chunking
 → embeddings
 → vector DB
 → retrieval
 → prompt
 → LLM
```