# Getting started with LangChain + OpenAI (Quickstart)

This document bundles a complete quickstart guide for the code you provided and explains every concept used: environment setup, key LangChain components (prompts, models, output parsers), vector stores and retrieval chains, LangSmith tracing, LangServe, Ollama integration, and a simple Streamlit demo. At the end there are troubleshooting tips and suggestions for exporting and deploying the example.

## Table of contents

1. Summary / Purpose
2. Requirements
3. Environment variables and tracing (LangSmith)
4. Installing packages and virtualenv
5. Core LangChain concepts used
   * LLMs
   * Prompts and ChatPromptTemplate
   * Chains (composition operator |)
   * Output parsers
   * Documents and text splitters
   * Embeddings and vector stores (FAISS)
   * Retrievers and retrieval-augmented generation (RAG)
6. Code walkthrough (cleaned and explained)
   * OpenAI Chat LLM example
   * Prompt template + chain
   * Loading website docs via WebBaseLoader
   * Text splitting
   * Embeddings + FAISS vector store
   * Retriever + Retrieval chain
   * Ollama + Streamlit example
7. Example requests and expected behavior
8. LangSmith tracing and what you see in the UI
9. LangServe basics (how to serve the chain)
10. Deployment & running locally (Streamlit example)
11. Troubleshooting & tips
12. Next steps and references

## 1. Summary / Purpose

You want a single coherent document that explains the example code and concepts so you can run and iterate quickly. This guide clarifies how each piece fits together and gives runnable examples.

## 2. Requirements

* Python 3.10+ recommended
* pip available
* A virtual environment (optional but recommended)
* API keys: OpenAI API key, LangChain/LangSmith API key (for tracing), Hugging Face token (if using HF models or connectors)
* The following Python packages (example):

langchain  
langchain-openai  
langchain-ollama  
langchain-community  
faiss-cpu  
python-dotenv  
streamlit

(Your requirements.txt should list the exact versions you tested.)

## 3. Environment variables and tracing (LangSmith)

Set your secrets in a .env file or directly in the environment. Example .env entries:

Open\_API\_Key=sk-... # OpenAI API key  
Langchain\_api\_key=... # LangSmith / LangChain tracing key  
Langchain\_project=MyProject # project name used in tracing  
HF\_token=hf\_... # Hugging Face token (optional)

Load them in your script using python-dotenv and export to os.environ:

from dotenv import load\_dotenv  
import os  
load\_dotenv()  
os.environ['OPENAI\_API\_KEY'] = os.getenv('Open\_API\_Key')  
os.environ['Langchain\_api\_key'] = os.getenv('Langchain\_api\_key')  
os.environ['LANGCHAIN\_TRACING\_V2'] = 'true'  
os.environ['Langchain\_project'] = os.getenv('Langchain\_project')  
os.environ['HF\_token'] = os.getenv('HF\_token')

Setting LANGCHAIN\_TRACING\_V2=true enables LangChain/ LangSmith tracing so that your run metadata, prompts and model responses are visible in LangSmith.

## 4. Installing packages and virtualenv

Example quick commands:

python -m venv venv  
source venv/bin/activate # macOS / Linux  
# venv\Scripts\activate # Windows  
pip install -r requirements.txt

You included ! pip install -r requirement.txt in notebooks — that is fine inside Jupyter with the ! magic.

## 5. Core LangChain concepts used

### LLMs

An LLM object (e.g., ChatOpenAI or OllamaLLM) is the interface to call an underlying model. It handles sending prompts, receiving responses, and exposing configuration like temperature, max\_tokens, and the model name.

### Prompts and ChatPromptTemplate

A ChatPromptTemplate lets you compose system and user messages with placeholders (e.g., {input} or {question}) to produce the final chat-style prompt.

### Chains

Chains compose components (prompt → model → parser) using the | operator in the langchain\_core style. Chains make small building blocks reusable.

### Output parsers

StrOutputParser (or custom parsers) transform the raw model output into structured Python objects (string, JSON, or a dataclass).

### Documents & Text splitters

Documents are the unit stored in vector DBs. RecursiveCharacterTextSplitter chunks long text into overlapping segments (helpful for context windows).

### Embeddings & Vector stores

Embeddings convert text into vectors. FAISS is an in-memory vector index used in examples for similarity search.

### Retriever & Retrieval Chains

A retriever wraps a vectorstore and returns the most relevant document chunks for a query. A retrieval chain feeds those chunks plus the question into an LLM to produce an answer grounded in the retrieved context.

## 6. Code walkthrough (cleaned examples + explanations)

The following are cleaned and annotated versions of your snippets.

### 6.1 OpenAI Chat LLM example

from langchain\_openai import ChatOpenAI  
llm = ChatOpenAI(model='gpt-4o')  
print(llm)  
# invoke directly  
result = llm.invoke('what is gen ai?')  
print(result.content)

**Notes:** - ChatOpenAI is a wrapper over OpenAI chat API. invoke() returns an object containing .content with the text. - Use temperature and max\_output\_tokens to tune creativity and length.

### 6.2 ChatPromptTemplate + chain composition

from langchain\_core.prompts import ChatPromptTemplate  
prompt = ChatPromptTemplate.from\_messages([  
 ("system", "you are expert AI engineer. Provide answers based on the question"),  
 ("user", "{input}")  
])  
  
# Compose with the llm  
chain = prompt | llm  
result = chain.invoke({"input": "Can you tell me about LangSmith"})  
print(result)

**Notes:** - You can stack more messages (assistant messages, examples, or few-shot in system). - Compose more steps: prompt | llm | output\_parser.

### 6.3 Output parser

from langchain\_core.output\_parsers import StrOutputParser  
output\_parser = StrOutputParser()  
chain = prompt | llm | output\_parser  
response = chain.invoke({"input": "Can you tell me about LangSmith"})  
print(response)

StrOutputParser just normalizes the final text. For structured responses use a JSON or Pydantic parser.

### 6.4 Loading documents from a website

from langchain\_community.document\_loaders import WebBaseLoader  
loader = WebBaseLoader("https://docs.smith.langchain.com/")  
docs = loader.load()  
# docs is a list of Document objects (page\_content, metadata)

**Notes:** - For scraping, use the appropriate loader for the site format. Respect robots.txt and site terms.

### 6.5 Splitting documents

from langchain\_text\_splitters import RecursiveCharacterTextSplitter  
text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=20)  
documents = text\_splitter.split\_documents(docs)

**Notes:** - chunk\_size=50 is small — choose chunk sizes that match your LLM context window (e.g., 500-1500 tokens) and overlap ~10-20%.

### 6.6 Embeddings and FAISS

from langchain\_openai import OpenAIEmbeddings  
embeddings = OpenAIEmbeddings()  
  
from langchain\_community.vectorstores import FAISS  
vectorestore\_db = FAISS.from\_documents(documents, embeddings)

**Notes:** - OpenAIEmbeddings() will call OpenAI to compute vectors — this costs API credits per call. - FAISS stores vectors in memory (or can be persisted to disk with save/load).

### 6.7 Similarity search

query = "AI applications involve writing prompts to instruct the LLM on what to do."  
results = vectorestore\_db.similarity\_search(query)  
print(results[0].page\_content)

This returns the most similar document chunk(s).

### 6.8 Retrieval + Document chain

from langchain.prompts import ChatPromptTemplate  
from langchain.chains.combine\_documents import create\_stuff\_documents\_chain  
  
prompt = ChatPromptTemplate.from\_template(  
"""  
Answer the following question based only on the provided Context:  
<context>  
{context}  
</context>  
"""  
)  
  
from langchain\_openai import ChatOpenAI  
llm = ChatOpenAI(model='gpt-4o')  
document\_chain = create\_stuff\_documents\_chain(llm=llm, prompt=prompt)  
  
# call directly with documents  
from langchain\_core.documents import Document  
result = document\_chain.invoke({  
 "input": "Some question",  
 "context": [Document(page\_content="AI applications involve writing prompts...")]  
})

**Notes:** - create\_stuff\_documents\_chain concatenates retrieved docs into the prompt (“stuffing”) and calls the LLM. - For long context, prefer map\_reduce or refine chains to avoid overflowing the prompt.

### 6.9 Putting retriever and chain together

retriever = vectorestore\_db.as\_retriever()  
from langchain.chains import create\_retrieval\_chain  
retrieval\_chain = create\_retrieval\_chain(retriever, document\_chain)  
  
response = retrieval\_chain.invoke({  
 "input": "AI applications involve writing prompts to instruct the LLM on what to do."  
})  
print(response['answer'])  
print(response['context'])

This runs: query → retriever → documents → llm chain → answer.

### 6.10 Ollama + Streamlit example

import streamlit as st  
from langchain\_ollama import OllamaLLM  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_core.output\_parsers import StrOutputParser  
  
prompt = ChatPromptTemplate.from\_messages([  
 ("system", "you are helpful assistant. please respond to the questions asked"),  
 ("user", "Question:{question}")  
])  
  
st.title("Langchain Demo With Gemma:2b")  
input\_text = st.text\_input("What question do you have in mind?")  
llm = OllamaLLM("gemma:2b")  
output\_parser = StrOutputParser()  
chain = prompt | llm | output\_parser  
  
if input\_text:  
 st.write(chain.invoke({"question": input\_text}))

**Notes:** - OllamaLLM expects a local Ollama server running with the model gemma:2b downloaded. - Ollama is a local-first LLM runtime; it doesn’t use OpenAI keys.

## 7. Example requests and expected behavior

* similarity\_search(query) returns top document chunks relevant to the query.
* retrieval\_chain.invoke(...) returns a dict with keys like answer and context depending on chain implementation.

## 8. LangSmith tracing and what you see in the UI

With tracing enabled you can view: - Prompt versions and the exact input sent to the model - Model responses and latency - Metadata (project, run name)

LangSmith records runs so you can debug prompt changes and reproduce failures.

## 9. LangServe basics (how to serve the chain)

LangServe allows exposing chains as APIs. Typical flow: 1. Create a FastAPI wrapper for the chain. 2. Add input validation (Pydantic). 3. Containerize (Docker) and deploy.

A minimal FastAPI endpoint example:

from fastapi import FastAPI  
app = FastAPI()  
  
@app.post('/query')  
async def query(payload: dict):  
 return retrieval\_chain.invoke({"input": payload['question']})

## 10. Deployment & running locally (Streamlit example)

Run the Streamlit demo:

streamlit run streamlit\_app.py

Open http://localhost:8501 and test the UI.

## 11. Troubleshooting & tips

* **API errors / rate limits**: check your keys and usage. Lower max\_tokens or batch fewer requests.
* **FAISS persistence**: save with vectorestore\_db.save\_local(path) and later FAISS.load\_local(path, embeddings).
* **Too long context**: reduce chunk size or switch chain type (map\_reduce).
* **Ollama connection**: ensure Ollama daemon is running and model is available.
* **Cost control**: cache embeddings and reuse vector indexes to avoid recomputing.

## 12. Next steps and references

* Try a map-reduce or refine chain for longer documents.
* Add a Pydantic schema for the API input and result.
* Persist FAISS to disk or use a production vector DB (Milvus, Weaviate, Pinecone).

*End of quickstart guide.*