🦜 🔗 The Complete LangChain x Gemini Masterclass

From Zero to Production-Ready AI Agents

Welcome! This guide is a complete curriculum designed to take you from a beginner to building complex, autonomous AI agents using **Google Gemini**. We will cover the theory, the code, and the "why" behind every component.

📚 Table of Contents

Part 1: The Big Picture (Theory)

What is LangChain?

LangChain is a **framework for orchestration**.

- **Raw APIs (like Gemini's)** take a string and give a string.
- **LangChain** lets you build *systems* around that model: connecting it to data (RAG), giving it memory, or allowing it to use tools (Agents).

Key Concepts

1. **Chains**: Linking steps together (Input → Prompt → Model → Output).
2. **Agents**: Systems where the LLM decides *which* steps to take next (Reasoning).
3. **LCEL (LangChain Expression Language)**: A standard syntax (using `|`) to compose these chains easily.

Part 2: Setup

We need the modern LangChain ecosystem.

Installation

```
pip install -qU langchain langchain-google-genai langchain-community langchain-cc
```

Authentication

We use `ChatGoogleGenerativeAI` to access Gemini.

```
import os
import getpass

# 1. Set API Key (Get from aistudio.google.com)
if "GOOGLE_API_KEY" not in os.environ:
    os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter Google API Key: ")

from langchain_google_genai import ChatGoogleGenerativeAI

# 2. Initialize Model
# We use "flash" for speed/cost, "pro" for complex reasoning.
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",
    temperature=0.7
)
```

Module 1: Model I/O (The Basics)

Theory: An LLM is a text-in, text-out engine. To make it useful, we need to structure the **Input** (Prompt Templates) and the **Output** (Parsers).

1.1 Prompt Templates

Don't use f-strings; use Templates. They allow you to swap variables easily and manage system instructions.

```
from langchain_core.prompts import ChatPromptTemplate

# Definition
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant specialized in {topic}."),
    ("human", "{question}"),
])

# Usage (Logic is separated from data)
prompt_value = prompt_template.invoke({"topic": "Python", "question": "What is a
```

1.2 Output Parsers

Gemini returns a complex `AIMessage` object. Parsers strip this down to what you need (like a string or JSON).

```
from langchain_core.output_parsers import StrOutputParser

parser = StrOutputParser()
```

Module 2: LCEL, Streaming & Async

Theory: **LCEL** uses the pipe `|` operator (like in Unix) to pass the output of one component into the input of the next.

2.1 The Basic Chain

```
# The "Run" Protocol: Prompt -> Model -> Parser
chain = prompt_template | llm | parser

# Invoke
result = chain.invoke({"topic": "Space", "question": "How far is the moon?"})
print(result)
```

2.2 Streaming ("The Async Thingy")

Theory: LLMs are slow. Users hate waiting. **Streaming** delivers tokens one by one (like a typewriter effect).

```
print("Streamed Answer: ", end="")
for chunk in chain.stream({"topic": "Oceans", "question": "How deep is the Marian
    print(chunk, end="", flush=True)
```

2.3 Async (Concurrency)

Theory: If you need to process 100 emails, doing them one-by-one is slow. `aiohttp` and Async methods allow you to do them all at once.

```
import asyncio

async def process_batch():
    inputs = [
        {"topic": "Math", "question": "What is 2+2?"},
        {"topic": "History", "question": "Who was Napoleon?"}
    ]
    # .abatch runs in parallel!
    results = await chain.abatch(inputs)
    print(results)

# await process_batch() # Run this in Jupyter/Python script
```

Module 3: Data & RAG (Retrieval Augmented Generation)

Theory: LLMs hallucinate. **RAG** fixes this by:

1. **Loading** your real data (PDFs, Websites).
2. **Splitting** it into small chunks.
3. **Embedding** it (converting text to number vectors).
4. **Retrieving** the right chunk when you ask a question.

3.1 Document Loaders & Splitters

```
from langchain_community.document_loaders import WebBaseLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

# 1. Load Data
loader = WebBaseLoader("[https://docs.langchain.com/oss/python/learn](https://doc
docs = loader.load()

# 2. Split Data (Chunks of 1000 chars)
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
splits = splitter.split_documents(docs)
```

3.2 Vector Database & Retrieval

We use Gemini's Embedding model to understand the *meaning* of the text.

```
from langchain_google_genai import GoogleGenerativeAIEmbeddings
from langchain_community.vectorstores import InMemoryVectorStore

# 1. Embed & Store
embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")
vectorstore = InMemoryVectorStore.from_documents(splits, embeddings)

# 2. Create Retriever (The search engine)
retriever = vectorstore.as_retriever()
```

3.3 The RAG Chain

```
from langchain_core.runnables import RunnablePassthrough

rag_prompt = ChatPromptTemplate.from_template("""
Answer based ONLY on this context:
{context}

Question: {question}
""")

rag_chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | rag_prompt
```

```
    | llm
    | parser
)

print(rag_chain.invoke("What is a LangChain Agent?"))
```

Module 4: Agents & Tools (The Brain)

Theory: A standard Chain is hardcoded. An **Agent** uses a loop called **ReAct** (Reason + Act).

1. **Thought**: "The user asked for weather. I need to check my tools."
2. **Action**: Call `get_weather("New York")`.
3. **Observation**: "It is sunny."
4. **Answer**: "It's sunny in New York!"

4.1 The `@tool` Decorator

This makes Python functions understandable by Gemini.

```
from langchain_core.tools import tool

@tool
def get_weather(city: str) -> str:
    """Get the current weather for a specific city. Call this if user asks about
    # In a real app, you'd call a weather API here.
    return f"The weather in {city} is sunny and 25°C."

@tool
def web_search(query: str) -> str:
    """Search the internet for current events."""
    return f"I found some results for {query}..."

tools = [get_weather, web_search]

# Bind tools to the LLM (Give Gemini "hands")
llm_with_tools = llm.bind_tools(tools)
```

Module 5: Memory (State)

Theory: HTTP and LLMs are stateless. If you say "Hi", then "My name is Bob", the model forgets "Hi". We must manually inject **History** into every new prompt.

```
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory

# 1. Create a chain that accepts history
history_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    ("placeholder", "{chat_history}"), # <--- The Slot for Memory
    ("human", "{input}"),
```

```
])
history_chain = history_prompt | llm | parser

# 2. Setup Persistence (In-Memory for now)
store = {}

def get_session_history(session_id: str):
    if session_id not in store:
        store[session_id] = ChatMessageHistory()
    return store[session_id]

# 3. Wrap the Chain
conversation = RunnableWithMessageHistory(
    history_chain,
    get_session_history,
    input_messages_key="input",
    history_messages_key="chat_history",
)

# 4. Chat!
config = {"configurable": {"session_id": "user_1"}}
conversation.invoke({"input": "Hi! I'm Bob."}, config=config)
conversation.invoke({"input": "What is my name?"}, config=config)
# Output: "Your name is Bob."
```

Module 6: LangGraph (Production Agents)

Theory: The old `AgentExecutor` was too rigid. **LangGraph** treats agents as a **State Machine** (Graph). It allows loops, conditional logic, and "Human-in-the-loop" checks.

6.1 Building the Graph

We will build a simple "ReAct" style agent graph.

```
from langgraph.graph import StateGraph, START, END
from langgraph.prebuilt import ToolNode, tools_condition
from typing import Annotated, TypedDict
from langgraph.graph.message import add_messages
from langchain_core.messages import BaseMessage

# 1. Define State (The "Memory" of the graph)
class AgentState(TypedDict):
    messages: Annotated[list[BaseMessage], add_messages]

# 2. Define Nodes
def chatbot(state: AgentState):
    # The LLM decides: Tool call? Or Final Answer?
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

# 3. Build Graph
graph_builder = StateGraph(AgentState)

graph_builder.add_node("chatbot", chatbot)
graph_builder.add_node("tools", ToolNode(tools=tools))

# 4. Define Edges (The Logic Flow)
graph_builder.add_edge(START, "chatbot")
```

```
# Conditional: If LLM asks for tool -> Go to "tools". Else -> End.
graph_builder.add_conditional_edges("chatbot", tools_condition)

# Loop: After tool runs, go back to chatbot to read the result.
graph_builder.add_edge("tools", "chatbot")

# 5. Compile
agent = graph_builder.compile()
```

6.2 Running the Agent

```
# User asks a question requiring a tool
final_state = agent.invoke({"messages": [("user", "What is the weather in London?

# Print the conversation flow
for msg in final_state["messages"]:
    print(f"[{msg.type.upper()}]: {msg.content}")
```

Summary: What have we learned?

1. **Chains**: The `|` operator is your best friend for clean code.
2. **RAG**: Never trust an LLM with facts unless you give it the docs via Vector Retrieval.
3. **Agents**: ReAct (Reason+Act) is the pattern where models use Tools (`@tool`).
4. **LangGraph**: The modern way to build loop-based agents that are robust enough for