📖 The Ultimate LangChain x Gemini Manual

The Complete Guide to Building Production AI Agents

Version: 2025 Edition **Engine:** Google Gemini 1.5 Flash / Pro **Frameworks:** LangChain (Core), LangGraph (Agents)

📋 Table of Contents

🛠️ Chapter 0: The Modern Stack

We strictly use the modern packages. Avoid legacy `langchain.agents` or `AgentExecutor`.

0.1 Installation

```
pip install -U langchain langchain-google-genai langchain-community langchain-cor
```

0.2 Authentication

Set your Google API key once at the start of your application.

```
import os
import getpass
from langchain_google_genai import ChatGoogleGenerativeAI

if "GOOGLE_API_KEY" not in os.environ:
    os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter Google API Key: ")

# Global Model Instance
# 'flash' is fast/cheap. 'pro' is smarter.
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",
    temperature=0,
    max_retries=2,
)
```

🧱 Chapter 1: LangChain Core

1.1 Structured Output (The Most Important Feature)

LLMs output text by default. You want Objects. Gemini supports this natively.

```
from pydantic import BaseModel, Field

# 1. Define the Schema
class WeatherResponse(BaseModel):
    temperature: float = Field(description="The temperature in Celsius")
    conditions: str = Field(description="e.g., 'Sunny', 'Raining'")
    advice: str = Field(description="Clothing advice based on weather")

# 2. Bind Schema to Model
structured_llm = llm.with_structured_output(WeatherResponse)

# 3. Invoke
result = structured_llm.invoke("It's scorching hot in Dubai today, about 40 degre
print(f"Advice: {result.advice}") # Output: "Wear light clothes..."
```

1.2 LCEL & Parallel Chains

Use `RunnableParallel` to run multiple prompts at once (saving time).

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableParallel

chain_summary = ChatPromptTemplate.from_template("Summarize: {topic}") | llm
chain_joke = ChatPromptTemplate.from_template("Tell a joke about: {topic}") | llm

# Runs BOTH in parallel
map_chain = RunnableParallel(summary=chain_summary, joke=chain_joke)

result = map_chain.invoke({"topic": "Quantum Physics"})
print("Summary:", result['summary'].content)
print("Joke:", result['joke'].content)
```

📚 Chapter 2: Retrieval Augmented Generation (RAG)

2.1 The Standard Pipeline

Load -> Split -> Embed -> Store -> Retrieve.

```
from langchain_community.document_loaders import WebBaseLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_google_genai import GoogleGenerativeAIEmbeddings
from langchain_community.vectorstores import InMemoryVectorStore

# 1. Load & Split
docs = WebBaseLoader("[https://docs.langchain.com/oss/python/learn](https://docs.
```

```
splits = RecursiveCharacterTextSplitter(chunk_size=1000).split_documents(docs)

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

2.2 History-Aware Retrieval (Advanced)

If a user says "How does it work?", the LLM needs to know what "it" is. This chain rewrites the question first.

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

# A. Rewrite Prompt
rewrite_prompt = ChatPromptTemplate.from_messages([
    ("system", "Rewrite the user question to be standalone using the chat history
    ("placeholder", "{chat_history}"),
    ("human", "{input}"),
])

# B. The Rewriter Chain
history_aware_retriever = rewrite_prompt | llm | StrOutputParser() | retriever

# C. The Answer Chain
qa_prompt = ChatPromptTemplate.from_messages([
    ("system", "Answer using ONLY this context:\n{context}"),
    ("placeholder", "{chat_history}"),
    ("human", "{input}"),
])

# D. Full Pipeline
rag_chain = (
    RunnablePassthrough.assign(
        context=lambda x: "\n".join([d.page_content for d in history_aware_retrie
    )
    | qa_prompt
    | llm
    | StrOutputParser()
)
```

Chapter 3: LangGraph Basics

LangGraph replaces `AgentExecutor`. It models agents as a Graph (Nodes + Edges).

3.1 State & Tools

The **State** is the shared memory of the agent.

```
from typing import Annotated, TypedDict
from langgraph.graph import StateGraph, START, END
```

```
from langgraph.graph.message import add_messages
from langchain_core.messages import BaseMessage, HumanMessage
from langchain_core.tools import tool

# 1. Define State (Messages are appended, not overwritten)
class State(TypedDict):
    messages: Annotated[list[BaseMessage], add_messages]

# 2. Define Tools
@tool
def web_search(query: str):
    """Simulates a search engine."""
    return f"Results for {query}: LangChain is cool."

tools = [web_search]
llm_with_tools = llm.bind_tools(tools)
```

3.2 Nodes & Logic

Nodes are functions that do work.

```
from langgraph.prebuilt import ToolNode, tools_condition

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

# Prebuilt node to run tools
tool_node = ToolNode(tools)
```

3.3 Wiring the Graph

```
builder = StateGraph(State)

# Add Nodes
builder.add_node("chatbot", chatbot)
builder.add_node("tools", tool_node)

# Add Edges
builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition) # Go to 'tools' or END
builder.add_edge("tools", "chatbot") # Loop back!

# Compile
graph = builder.compile()
```

🚦 Chapter 4: Production Controls

This is what separates a "Toy" from a "Product".

4.1 Persistence (Long-term Memory)

Use a **Checkpointer** to save state to a database (using `MemorySaver` here for demo).

```
from langgraph.checkpoint.memory import MemorySaver

memory = MemorySaver()
graph_persistent = builder.compile(checkpointer=memory)

# Thread ID acts as the Session ID
config = {"configurable": {"thread_id": "user_123"}}
```

4.2 Human-in-the-Loop (Approval)

Pause the agent before it runs a tool.

```
# Pause BEFORE entering the 'tools' node
graph_hitl = builder.compile(checkpointer=memory, interrupt_before=["tools"])

# 1. Run (Agent stops before tool)
graph_hitl.invoke(
    {"messages": [HumanMessage(content="Search for LangGraph")]},
    config=config
)

# 2. Inspect
snapshot = graph_hitl.get_state(config)
print("Next Step:", snapshot.next) # ('tools',)

# 3. Resume (Approve)
graph_hitl.invoke(None, config=config)
```

4.3 Time Travel (State Updates)

Edit the conversation while the agent is paused.

```
# While paused...
graph_hitl.update_state(
    config,
    {"messages": [HumanMessage(content="Search for LangGraph 2.0 instead")]}
)
# Now resume, and it processes the NEW instruction.
```

🤖 Chapter 5: Multi-Agent Systems

The **Supervisor Pattern**: A router agent delegates tasks to workers.

```
from typing import Literal

# 1. The Supervisor (Router)
class Router(BaseModel):
    next: Literal["Researcher", "Coder", "FINISH"]
```

```
supervisor_prompt = ChatPromptTemplate.from_template(
    "You are a manager. Select the next worker: Researcher, Coder, or FINISH."
)
supervisor_chain = supervisor_prompt | llm.with_structured_output(Router)

# 2. The Workers
def researcher(state):
    return {"messages": [HumanMessage(content="I found the docs.", name="Research

def coder(state):
    return {"messages": [HumanMessage(content="I wrote the code.", name="Coder")]

def supervisor(state):
    result = supervisor_chain.invoke(state)
    return {"next": result.next}

# 3. The Graph
class MultiAgentState(State):
    next: str

ma_builder = StateGraph(MultiAgentState)
ma_builder.add_node("Researcher", researcher)
ma_builder.add_node("Coder", coder)
ma_builder.add_node("Supervisor", supervisor)

ma_builder.add_edge(START, "Supervisor")
ma_builder.add_edge("Researcher", "Supervisor")
ma_builder.add_edge("Coder", "Supervisor")
ma_builder.add_conditional_edges("Supervisor", lambda x: x["next"])

ma_graph = ma_builder.compile()
```

🔍 Appendix: Debugging

LangSmith

To see exactly what is happening inside your chains (latency, inputs, outputs, errors), simply set these variables in your environment. No code changes needed.

```
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = "your-langchain-api-key"
os.environ["LANGCHAIN_PROJECT"] = "Gemini Masterclass"
```