Reference Card: LangChain
Yogesh Haribhau Kulkarni

Installation & Imports

```
# Installation
pip install langchain langchain−openai langchain−community
pip install langchain−anthropic langchain−google−genai

# Core Imports
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage, SystemMessage, AIMessage
from langchain.prompts import PromptTemplate, ChatPromptTemplate
from langchain.chains import LLMChain, SimpleSequentialChain
from langchain.memory import ConversationBufferMemory
from langchain.agents import initialize_agent, Tool, AgentType
```

LLM Models - Basic Setup

```
# OpenAI LLM (Legacy)
from langchain.llms import OpenAI
llm = OpenAI(temperature=0.7, model_name=”gpt−3.5−turbo−instruct”)
response = llm(”What is LangChain?”)

# Chat Models (Recommended)
from langchain.chat_models import ChatOpenAI
chat = ChatOpenAI(temperature=0.7, model_name=”gpt−4”)
result = chat.invoke(”Explain quantum computing”)

# Anthropic Claude
from langchain_anthropic import ChatAnthropic
claude = ChatAnthropic(model=”claude−3−sonnet−20240229”)

# Google Gemini
from langchain_google_genai import ChatGoogleGenerativeAI
gemini = ChatGoogleGenerativeAI(model=”gemini−pro”)
```

Message Types & Chat

```
# Message Schema
from langchain.schema import HumanMessage, SystemMessage, AIMessage

messages = [
    SystemMessage(content=”You are a helpful AI assistant”),
    HumanMessage(content=”What is machine learning?”),
    AIMessage(content=”ML is a subset of AI...”),
    HumanMessage(content=”Give me an example”)
]

# Invoke Chat with Messages
chat = ChatOpenAI()
response = chat.invoke(messages)
print(response.content)

# Batch Processing
responses = chat.batch([
    [HumanMessage(content=”Hi”)],
    [HumanMessage(content=”Bye”)]
])

# Streaming
for chunk in chat.stream(”Tell me a story”):
    print(chunk.content, end=””, flush=True)
```

Prompt Templates - Basic

```
# Simple Prompt Template
from langchain.prompts import PromptTemplate

template = ”Tell me a {adjective} joke about {content}”
prompt = PromptTemplate(
    input_variables=[”adjective”, ”content”],
    template=template
)
formatted = prompt.format(adjective=”funny”, content=”programmers”)

# Chat Prompt Template
from langchain.prompts import ChatPromptTemplate

chat_template = ChatPromptTemplate.from_messages([
    (”system”, ”You are a helpful assistant that translates {input_lang} to {output_lang}”),
    (”human”, ”{text}”)
])
messages = chat_template.format_messages(
    input_lang=”English”, output_lang=”French”, text=”Hello”
)

# Few−Shot Prompts
from langchain.prompts import FewShotPromptTemplate
examples = [{”word”: ”happy”, ”antonym”: ”sad”}]
```

Chains - LLMChain

```
# Basic LLMChain
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

llm = ChatOpenAI()
prompt = PromptTemplate(
    input_variables=[”product”],
    template=”What is a good name for a company that makes {product}?”
)
chain = LLMChain(llm=llm, prompt=prompt)
result = chain.run(product=”eco−friendly shoes”)

# Multiple Inputs
chain.run({”product”: ”AI software”, ”style”: ”creative”})

# Invoke Method (New)
result = chain.invoke({”product”: ”bikes”})
print(result[”text”])

# Chain with Output Parser
from langchain.output_parsers import CommaSeparatedListOutputParser
parser = CommaSeparatedListOutputParser()
chain = LLMChain(llm=llm, prompt=prompt, output_parser=parser)
```

Sequential Chains

```
# SimpleSequentialChain − Single Input/Output
from langchain.chains import SimpleSequentialChain

chain1 = LLMChain(llm=llm, prompt=prompt1)  # Generate synopsis
chain2 = LLMChain(llm=llm, prompt=prompt2)  # Generate review
overall_chain = SimpleSequentialChain(chains=[chain1, chain2])
review = overall_chain.run(”tragedy”)

# SequentialChain − Multiple Inputs/Outputs
from langchain.chains import SequentialChain

chain = SequentialChain(
    chains=[chain1, chain2, chain3],
    input_variables=[”era”, ”genre”],
    output_variables=[”synopsis”, ”review”, ”social_post”],
    verbose=True
)
result = chain({”era”: ”medieval”, ”genre”: ”fantasy”})

# Access Outputs
print(result[”synopsis”])
print(result[”review”])
```

LCEL - LangChain Expression Language

```
# Basic LCEL Chain
from langchain.schema.runnable import RunnablePassthrough

prompt = ChatPromptTemplate.from_template("Tell me about {topic}")
chain = prompt | llm
result = chain.invoke({"topic": "neural networks"})

# Chain with Output Parser
from langchain.schema.output_parser import StrOutputParser
chain = prompt | llm | StrOutputParser()
output = chain.invoke({"topic": "blockchain"})

# Parallel Execution
from langchain.schema.runnable import RunnableParallel
chain = RunnableParallel(
    joke=joke_chain,
    poem=poem_chain
)

# Branching
from langchain.schema.runnable import RunnableBranch
branch = RunnableBranch(
    (lambda x: "python" in x["topic"].lower(), python_chain),
    (lambda x: "java" in x["topic"].lower(), java_chain),
    default_chain
)
```

Memory - Conversation

```
# ConversationBufferMemory
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
memory.save_context({"input": "hi"}, {"output": "hello"})
print(memory.load_memory_variables({}))

# With Chain
chain = LLMChain(llm=llm, prompt=prompt, memory=memory)
chain.run("What's my name?")  # Remembers context

# ConversationBufferWindowMemory (Last K interactions)
from langchain.memory import ConversationBufferWindowMemory
memory = ConversationBufferWindowMemory(k=2)

# ConversationSummaryMemory
from langchain.memory import ConversationSummaryMemory
memory = ConversationSummaryMemory(llm=llm)

# ConversationTokenBufferMemory
from langchain.memory import ConversationTokenBufferMemory
memory = ConversationTokenBufferMemory(llm=llm, max_token_limit=100)
```

Document Loaders

```
# Text File Loader
from langchain.document_loaders import TextLoader
loader = TextLoader("data.txt")
docs = loader.load()

# PDF Loader
from langchain.document_loaders import PyPDFLoader
loader = PyPDFLoader("document.pdf")
pages = loader.load_and_split()

# CSV Loader
from langchain.document_loaders import CSVLoader
loader = CSVLoader("data.csv")

# Web Page Loader
from langchain.document_loaders import WebBaseLoader
loader = WebBaseLoader("https://example.com")

# Directory Loader
from langchain.document_loaders import DirectoryLoader
loader = DirectoryLoader("./docs", glob="**/*.txt")

# Unstructured Loader (Multiple formats)
from langchain.document_loaders import UnstructuredFileLoader
loader = UnstructuredFileLoader("file.docx")
```

Text Splitters

```
# Character Text Splitter
from langchain.text_splitter import CharacterTextSplitter

splitter = CharacterTextSplitter(
    separator="\n\n",
    chunk_size=1000,
    chunk_overlap=200
)
texts = splitter.split_text(long_text)

# Recursive Character Text Splitter (Recommended)
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
    separators=["\n\n", "\n", " ", ""]
)
chunks = splitter.split_documents(documents)

# Token Text Splitter
from langchain.text_splitter import TokenTextSplitter
splitter = TokenTextSplitter(chunk_size=100, chunk_overlap=10)

# Markdown Splitter
from langchain.text_splitter import MarkdownTextSplitter
```

Embeddings & Vector Stores

```
# OpenAI Embeddings
from langchain.embeddings import OpenAIEmbeddings
embeddings = OpenAIEmbeddings()
vector = embeddings.embed_query("Hello world")
doc_vectors = embeddings.embed_documents(["doc1", "doc2"])

# HuggingFace Embeddings
from langchain.embeddings import HuggingFaceEmbeddings
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# FAISS Vector Store
from langchain.vectorstores import FAISS
vectorstore = FAISS.from_documents(documents, embeddings)
vectorstore.save_local("faiss_index")
loaded_vs = FAISS.load_local("faiss_index", embeddings)

# Chroma Vector Store
from langchain.vectorstores import Chroma
vectorstore = Chroma.from_documents(documents, embeddings, persist_directory="./chroma_db")

# Pinecone
from langchain.vectorstores import Pinecone
vectorstore = Pinecone.from_documents(documents, embeddings, index_name="myindex")
```

Retrieval - Similarity Search

```
# Basic Similarity Search
docs = vectorstore.similarity_search("query text", k=4)

# Similarity Search with Scores
docs_scores = vectorstore.similarity_search_with_score("query", k=3)
for doc, score in docs_scores:
    print(f"Score: {score}, Content: {doc.page_content}")

# Max Marginal Relevance (MMR) - Diverse Results
docs = vectorstore.max_marginal_relevance_search("query", k=4, fetch_k=20)

# As Retriever
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 3}
)
docs = retriever.get_relevant_documents("query")

# MMR Retriever
retriever = vectorstore.as_retriever(
    search_type="mmr",
    search_kwargs={"k": 3, "fetch_k": 10}
)

# Threshold-based
retriever = vectorstore.as_retriever(search_kwargs={"score_threshold": 0.5})
```

RAG - Retrieval Augmented Generation

```
# RetrievalQA Chain
from langchain.chains import RetrievalQA

qa = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=retriever,
    return_source_documents=True
)
result = qa({"query": "What is the main topic?"})
print(result["result"])
print(result["source_documents"])

# Chain Types: stuff, map_reduce, refine, map_rerank

# ConversationalRetrievalChain
from langchain.chains import ConversationalRetrievalChain

qa = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=retriever,
    memory=memory
)
result = qa({"question": "What is X?", "chat_history": []})

# LCEL RAG Chain
from langchain.schema.runnable import RunnablePassthrough
chain = {"context": retriever, "question": RunnablePassthrough()} | prompt | llm
```

Agents - Basic Setup

```
# Define Tools
from langchain.agents import Tool

def search_function(query):
    return f"Results for: {query}"

tools = [
    Tool(
        name="Search",
        func=search_function,
        description="useful for searching information"
    ),
    Tool(
        name="Calculator",
        func=lambda x: eval(x),
        description="useful for math calculations"
    )
]

# Initialize Agent
from langchain.agents import initialize_agent, AgentType

agent = initialize_agent(
    tools=tools,
    llm=llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)

# Run Agent
result = agent.run("What is 25 * 4?")
```

Agent Types & Tools

```
# Agent Types
# ZERO_SHOT_REACT_DESCRIPTION - Best for most cases
# CONVERSATIONAL_REACT_DESCRIPTION - With memory
# REACT_DOCSTORE - For document reasoning
# SELF_ASK_WITH_SEARCH - For factual questions
# OPENAI_FUNCTIONS - Uses OpenAI function calling

# Built-in Tools
from langchain.agents import load_tools

tools = load_tools(["serpapi", "llm-math"], llm=llm)
tools = load_tools(["wikipedia", "arxiv"])

# Custom Tool with Decorator
from langchain.tools import tool

@tool
def get_weather(location: str) -> str:
    """Get weather for a location"""
    return f"Weather in {location}: Sunny, 72F"

# Tool from Function
from langchain.tools import StructuredTool
tool = StructuredTool.from_function(
    func=my_function,
    name="MyTool",
    description="Does something"
)
```

OpenAI Functions Agent

```
# OpenAI Functions Agent (Recommended)
from langchain.agents import create_openai_functions_agent
from langchain.agents import AgentExecutor

tools = [search_tool, calculator_tool]
llm = ChatOpenAI(model="gpt-4", temperature=0)

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant"),
    ("user", "{input}"),
    ("assistant", "{agent_scratchpad}")
])

agent = create_openai_functions_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

result = agent_executor.invoke({"input": "Search for Python tutorials"})

# Streaming
for chunk in agent_executor.stream({"input": "Calculate 100/5"}):
    print(chunk)

# With Memory
memory = ConversationBufferMemory(memory_key="chat_history")
agent_executor = AgentExecutor(agent=agent, tools=tools, memory=memory)
```

Output Parsers

```
# String Output Parser
from langchain.schema.output_parser import StrOutputParser
parser = StrOutputParser()
chain = prompt | llm | parser

# List Output Parser
from langchain.output_parsers import CommaSeparatedListOutputParser
parser = CommaSeparatedListOutputParser()
format_instructions = parser.get_format_instructions()

# Structured Output Parser
from langchain.output_parsers import StructuredOutputParser, ResponseSchema
schemas = [
    ResponseSchema(name="answer", description="answer to question"),
    ResponseSchema(name="source", description="source of info")
]
parser = StructuredOutputParser.from_response_schemas(schemas)
```

```
# Pydantic Output Parser
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field

class Person(BaseModel):
    name: str = Field(description="person's name")
    age: int = Field(description="person's age")

parser = PydanticOutputParser(pydantic_object=Person)
```

Callbacks & Tracing

```
# Custom Callback Handler
from langchain.callbacks.base import BaseCallbackHandler

class MyCallback(BaseCallbackHandler):
    def on_llm_start(self, serialized, prompts, **kwargs):
        print(f"LLM started with prompts: {prompts}")

    def on_llm_end(self, response, **kwargs):
        print(f"LLM ended with: {response}")

# Use Callback
handler = MyCallback()
llm = ChatOpenAI(callbacks=[handler])

# Streaming Callback
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
llm = ChatOpenAI(streaming=True, callbacks=[StreamingStdOutCallbackHandler()])

# LangSmith Tracing (Environment Variables)
import os
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = "your-api-key"

# Context Manager
from langchain.callbacks import tracing_enabled
with tracing_enabled():
    chain.run("query")
```

Caching

```
# In-Memory Cache
from langchain.cache import InMemoryCache
from langchain.globals import set_llm_cache

set_llm_cache(InMemoryCache())
llm = ChatOpenAI()
# First call - slow
llm.invoke("Tell me a joke")
# Second call - instant (cached)
llm.invoke("Tell me a joke")

# SQLite Cache
from langchain.cache import SQLiteCache
set_llm_cache(SQLiteCache(database_path=".langchain.db"))

# Redis Cache
from langchain.cache import RedisCache
import redis
set_llm_cache(RedisCache(redis_=redis.Redis()))

# Semantic Cache (Vector-based)
from langchain.cache import RedisSemanticCache
set_llm_cache(RedisSemanticCache(
    redis_url="redis://localhost:6379",
    embedding=embeddings
))
```

Router Chains

```
# MultiPromptChain - Route to Different Prompts
from langchain.chains.router import MultiPromptChain
from langchain.chains.router.llm_router import LLMRouterChain, RouterOutputParser

physics_template = "You are a physics expert..."
math_template = "You are a math expert..."

prompt_infos = [
    {"name": "physics", "description": "Good for physics questions", "prompt_template": physics_template},
    {"name": "math", "description": "Good for math questions", "prompt_template": math_template}
]

destinations = [f"{p['name']}: {p['description']}" for p in prompt_infos]
router_template = "Route to appropriate expert..."

router_chain = LLMRouterChain.from_llm(llm, router_prompt)
chain = MultiPromptChain(
    router_chain=router_chain,
    destination_chains=destination_chains,
    default_chain=default_chain
)

result = chain.run("What is Newton's law?")
```

SQL Database Chain

```
# SQL Database Integration
from langchain.utilities import SQLDatabase
from langchain.chains import SQLDatabaseChain

db = SQLDatabase.from_uri("sqlite:///mydb.db")
db_chain = SQLDatabaseChain.from_llm(llm, db, verbose=True)

# Natural Language to SQL
result = db_chain.run("How many users are in the database?")

# SQL Database Sequence Chain
from langchain.chains import SQLDatabaseSequentialChain
chain = SQLDatabaseSequentialChain.from_llm(llm, db)

# Custom SQL Query
from langchain.chains import create_sql_query_chain
chain = create_sql_query_chain(llm, db)
query = chain.invoke({"question": "List all products"})

# Execute Query
from langchain.tools import QuerySQLDataBaseTool
execute = QuerySQLDataBaseTool(db=db)
result = execute.run(query)
```

API Chains & Requests

```
# API Chain
from langchain.chains import APIChain
from langchain.chains.api import open_meteo_docs

chain = APIChain.from_llm_and_api_docs(
    llm,
    open_meteo_docs.OPEN_METEO_DOCS,
    verbose=True
)
result = chain.run("What's the weather in London?")

# LLM Requests Chain
from langchain.chains import LLMRequestsChain

template = """Extract info from {requests_result}"""
chain = LLMRequestsChain(llm_chain=LLMChain(llm=llm, prompt=prompt))
inputs = {"url": "https://api.example.com/data"}
result = chain(inputs)

# Custom API Tool
from langchain.tools import APIOperation
from langchain.utilities import RequestsWrapper

requests = RequestsWrapper()
tool = APIOperation.from_openapi_spec(spec, requests)
```

Evaluation & Testing

```
# Question Answering Evaluation
from langchain.evaluation.qa import QAEvalChain

examples = [
    {"query": "What is 2+2?", "answer": "4"},
    {"query": "Capital of France?", "answer": "Paris"}
]

predictions = chain.apply(examples)
eval_chain = QAEvalChain.from_llm(llm)
graded_outputs = eval_chain.evaluate(examples, predictions)

# String Evaluation
from langchain.evaluation import load_evaluator

evaluator = load_evaluator("criteria", criteria="conciseness")
result = evaluator.evaluate_strings(
    prediction="The answer is 4",
    reference="4",
    input="What is 2+2?"
)

# Custom Evaluator
from langchain.evaluation import StringEvaluator
class MyEvaluator(StringEvaluator):
    def _evaluate_strings(self, prediction, reference=None, input=None):
        return {"score": 0.9}
```

Advanced RAG Patterns

```
# Multi-Query Retriever
from langchain.retrievers.multi_query import MultiQueryRetriever
retriever = MultiQueryRetriever.from_llm(
    retriever=vectorstore.as_retriever(),
    llm=llm
)

# Contextual Compression
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor

compressor = LLMChainExtractor.from_llm(llm)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=retriever
)

# Ensemble Retriever (Hybrid Search)
from langchain.retrievers import EnsembleRetriever
ensemble = EnsembleRetriever(
    retrievers=[bm25_retriever, faiss_retriever],
    weights=[0.5, 0.5]
)

# Parent Document Retriever
from langchain.retrievers import ParentDocumentRetriever
from langchain.storage import InMemoryStore
store = InMemoryStore()
retriever = ParentDocumentRetriever(vectorstore=vs, docstore=store)
```

Graph & LangGraph Basics

```
# LangGraph - State-based Workflows
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated

class State(TypedDict):
    messages: list
    next_step: str

def node1(state):
    return {"next_step": "node2"}

def node2(state):
    return {"messages": state["messages"] + ["Done"]}

workflow = StateGraph(State)
workflow.add_node("start", node1)
workflow.add_node("process", node2)
workflow.add_edge("start", "process")
workflow.add_edge("process", END)
workflow.set_entry_point("start")

app = workflow.compile()
result = app.invoke({"messages": [], "next_step": "start"})

# Conditional Edges
def router(state):
    return "process" if state["next_step"] == "yes" else END
workflow.add_conditional_edges("start", router)
```

Streaming & Async

```
# Streaming Responses
for chunk in llm.stream("Write a poem"):
    print(chunk.content, end="", flush=True)

# Chain Streaming
chain = prompt | llm | StrOutputParser()
for token in chain.stream({"topic": "AI"}):
    print(token, end="")

# Async Invocation
import asyncio

async def run_chain():
    result = await chain.ainvoke({"input": "Hello"})
    return result

asyncio.run(run_chain())

# Async Batch
results = await chain.abatch([{"input": "Hi"}, {"input": "Bye"}])

# Async Streaming
async for chunk in chain.astream({"input": "Tell me"}):
    print(chunk)

# Agent Streaming
for step in agent_executor.stream({"input": "Query"}):
    print(step)
```

Best Practices & Tips

```
# 1. Use LCEL for Modern Chains
chain = prompt | llm | parser  # Recommended
# vs old LLMChain

# 2. Batch for Efficiency
results = chain.batch([input1, input2, input3])

# 3. Set Temperature Appropriately
llm = ChatOpenAI(temperature=0)  # Deterministic
llm = ChatOpenAI(temperature=0.7)  # Creative

# 4. Handle Errors Gracefully
try:
    result = chain.invoke(input)
except Exception as e:
    print(f"Error: {e}")

# 5. Use Streaming for Long Responses
for chunk in chain.stream(input):
    display(chunk)

# 6. Limit Context Window
splitter = RecursiveCharacterTextSplitter(chunk_size=1000)

# 7. Use Async for Concurrent Operations
await asyncio.gather(chain1.ainvoke(x), chain2.ainvoke(y))

# 8. Monitor with Callbacks
llm = ChatOpenAI(callbacks=[MyCallbackHandler()])

# 9. Cache Expensive Calls
```

```
set_llm_cache(InMemoryCache())

# 10. Version Control Prompts
prompt_v1 = PromptTemplate.from_file(”prompts/v1.txt”)
```

Custom Tools & Toolkits

```
# Custom Tool with Schema
from langchain.tools import BaseTool
from pydantic import BaseModel, Field

class SearchInput(BaseModel):
    query: str = Field(description=”search query”)
    limit: int = Field(default=5, description=”result limit”)

class CustomSearchTool(BaseTool):
    name = ”custom_search”
    description = ”Search for information”
    args_schema = SearchInput

    def _run(self, query: str, limit: int = 5):
        return f”Results for {query}, limit {limit}”

    async def _arun(self, query: str, limit: int = 5):
        return self._run(query, limit)

# Use Toolkits
from langchain.agents.agent_toolkits import create_python_agent
from langchain.tools import PythonREPLTool

agent = create_python_agent(llm=llm, tool=PythonREPLTool())

# JSON Agent
from langchain.agents.agent_toolkits import create_json_agent
agent = create_json_agent(llm=llm, toolkit=json_toolkit)
```

Multi-Modal Inputs

```
# Image Inputs with GPT-4V
from langchain.schema.messages import HumanMessage

llm = ChatOpenAI(model=”gpt-4-vision-preview”)
message = HumanMessage(
    content=[
        {”type”: ”text”, ”text”: ”What’s in this image?”},
        {”type”: ”image_url”, ”image_url”: {”url”: ”https://example.com/img.jpg”}}
    ]
)
response = llm.invoke([message])

# Base64 Image
import base64
with open(”image.jpg”, ”rb”) as f:
    image_data = base64.b64encode(f.read()).decode()

message = HumanMessage(
    content=[
        {”type”: ”text”, ”text”: ”Describe this”},
        {”type”: ”image_url”, ”image_url”: {”url”: f”data:image/jpeg;base64,{image_data}”}}
    ]
)

# Multiple Images
content = [
    {”type”: ”text”, ”text”: ”Compare these images”},
    {”type”: ”image_url”, ”image_url”: {”url”: url1}},
    {”type”: ”image_url”, ”image_url”: {”url”: url2}}
]
```

Guardrails & Moderation

```
# Constitutional Chain (Content Filtering)
from langchain.chains.constitutional_ai.base import ConstitutionalChain
from langchain.chains.constitutional_ai.models import ConstitutionalPrinciple
```

```
principles = [
    ConstitutionalPrinciple(
        name=”harmful”,
        critique_request=”Is this harmful?”,
        revision_request=”Rewrite to be harmless”
    )
]

constitutional_chain = ConstitutionalChain.from_llm(
    llm=llm,
    chain=qa_chain,
    constitutional_principles=principles
)

# OpenAI Moderation
from langchain.chains import OpenAIModerationChain

moderation = OpenAIModerationChain()
result = moderation.run(”Check this text”)

# Input Validation
def validate_input(user_input):
    if len(user_input) > 1000:
        raise ValueError(”Input too long”)
    return user_input
```

Fallbacks & Retry Logic

```
# Fallback Models
from langchain.schema.runnable import RunnableWithFallbacks

primary_llm = ChatOpenAI(model=”gpt-4”)
fallback_llm = ChatOpenAI(model=”gpt-3.5-turbo”)

chain = primary_llm.with_fallbacks([fallback_llm])
result = chain.invoke(”Complex query”)

# Multiple Fallbacks
chain = primary.with_fallbacks([fallback1, fallback2, fallback3])

# Retry Logic
from langchain.schema.runnable import RunnableRetry

chain_with_retry = chain.with_retry(
    stop_after_attempt=3,
    wait_exponential_jitter=True
)

# Custom Error Handling
from langchain.schema.runnable import RunnablePassthrough

def error_handler(error):
    print(f”Error occurred: {error}”)
    return {”error”: str(error)}

chain = chain.with_config(
    run_name=”my_chain”,
    max_concurrency=5
)
```

Metadata & Filtering

```
# Add Metadata to Documents
from langchain.schema import Document

docs = [
    Document(
        page_content=”Content 1”,
        metadata={”source”: ”file1.txt”, ”date”: ”2024-01-01”, ”category”: ”tech”}
    ),
    Document(
        page_content=”Content 2”,
        metadata={”source”: ”file2.txt”, ”date”: ”2024-01-02”, ”category”: ”science”}
    )
]

# Metadata Filtering in Vector Store
vectorstore = Chroma.from_documents(docs, embeddings)
```

```
# Filter by Metadata
results = vectorstore.similarity_search(
    "query",
    filter={"category": "tech"}
)

# Complex Filters
results = vectorstore.similarity_search(
    "query",
    filter={"date": {"$gte": "2024-01-01"}, "category": {"$in": ["tech", "science"]}}
)

# Self-Query Retriever
from langchain.retrievers.self_query.base import SelfQueryRetriever
retriever = SelfQueryRetriever.from_llm(llm, vectorstore, document_contents, metadata_field_info)
```

HuggingFace Integration

```
# HuggingFace Hub Models
from langchain.llms import HuggingFaceHub

llm = HuggingFaceHub(
    repo_id="google/flan-t5-large",
    model_kwargs={"temperature": 0.5, "max_length": 512}
)

# HuggingFace Pipeline (Local)
from langchain.llms import HuggingFacePipeline
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

model_id = "gpt2"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id)
pipe = pipeline("text-generation", model=model, tokenizer=tokenizer)

llm = HuggingFacePipeline(pipeline=pipe)

# HuggingFace Text Generation Inference
from langchain.llms import HuggingFaceTextGenInference
llm = HuggingFaceTextGenInference(
    inference_server_url="http://localhost:8080/",
    max_new_tokens=512
)
```

LLM Configuration & Parameters

```
# Temperature: Randomness (0=deterministic, 1=creative)
llm = ChatOpenAI(temperature=0.7)

# Max Tokens: Response length limit
llm = ChatOpenAI(max_tokens=500)

# Top P: Nucleus sampling (0.1=conservative, 1=diverse)
llm = ChatOpenAI(top_p=0.9)

# Frequency Penalty: Reduce repetition (-2.0 to 2.0)
llm = ChatOpenAI(frequency_penalty=0.5)

# Presence Penalty: Encourage new topics (-2.0 to 2.0)
llm = ChatOpenAI(presence_penalty=0.6)

# Stop Sequences
llm = ChatOpenAI(stop=["\n", "END"])

# Model Specific Parameters
llm = ChatOpenAI(
    model="gpt-4-turbo-preview",
    temperature=0.7,
    max_tokens=1000,
    timeout=60,
    max_retries=2,
    request_timeout=30
)

# Streaming
llm = ChatOpenAI(streaming=True, callbacks=[handler])
```

Prompt Engineering Patterns

```
# Few-Shot Learning
from langchain.prompts import FewShotPromptTemplate

examples = [
    {"input": "happy", "output": "sad"},
    {"input": "tall", "output": "short"}
]

example_prompt = PromptTemplate(
    input_variables=["input", "output"],
    template="Input: {input}\nOutput: {output}"
)

few_shot_prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix="Give the antonym:",
    suffix="Input: {adjective}\nOutput:",
    input_variables=["adjective"]
)

# Chain of Thought
cot_template = """Let's solve this step by step:
Question: {question}
Step 1: Identify key information
Step 2: Apply relevant concepts
Step 3: Calculate/reason
Answer:"""

# ReAct Pattern
react_template = """Thought: {thought}
Action: {action}
Observation: {observation}"""
```

Production Deployment Tips

```
# 1. Environment Variables
import os
os.environ["OPENAI_API_KEY"] = "your-key"
os.environ["LANGCHAIN_TRACING_V2"] = "true"

# 2. Rate Limiting
from langchain.llms import OpenAI
llm = OpenAI(request_timeout=30, max_retries=3)

# 3. Error Handling
try:
    result = chain.invoke(input)
except Exception as e:
    logger.error(f"Chain failed: {e}")
    result = fallback_response

# 4. Logging
import logging
logging.basicConfig(level=logging.INFO)

# 5. Monitoring
from langchain.callbacks import get_openai_callback
with get_openai_callback() as cb:
    result = chain.run("query")
    print(f"Tokens: {cb.total_tokens}, Cost: ${cb.total_cost}")

# 6. Async for Scale
async def process_batch(inputs):
    tasks = [chain.ainvoke(inp) for inp in inputs]
    return await asyncio.gather(*tasks)

# 7. Connection Pooling
llm = ChatOpenAI(max_concurrency=10)
```

Quick Reference - Common Patterns

```
# Basic Chat
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI()
response = llm.invoke("Hello")
```

```
# Simple Chain
chain = prompt | llm | parser
result = chain.invoke({”input”: ”data”})

# RAG Pipeline
docs = loader.load()
chunks = splitter.split_documents(docs)
vectorstore = FAISS.from_documents(chunks, embeddings)
retriever = vectorstore.as_retriever()
qa = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
answer = qa.run(”question”)
```

```
# Agent with Tools
tools = [search_tool, calculator_tool]
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION)
result = agent.run(”query”)

# Memory + Chain
memory = ConversationBufferMemory()
chain = LLMChain(llm=llm, prompt=prompt, memory=memory)
response = chain.run(”input”)
```

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it. If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.