

Reference Card: LangChain

Yogesh Haribhau Kulkarni

Introduction

Introduction to LangChain

What is LangChain?

A Framework for Building LLM Applications

Key Capabilities:

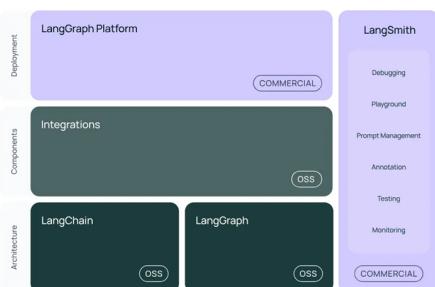
- Connect LLMs to external data (RAG)
- Build intelligent agents
- Manage prompts & memory
- Switch between models easily

Quick Facts:

- Open Source (MIT License)
- Python & JavaScript
- Created by Harrison Chase



From Idea to Production



(Ref: Getting Started with LangChain: A Beginner's Guide)

Why You Need LangChain

The Challenge:

- LLMs lack memory
- No access to your data
- Can't use external tools
- Complex integration code

The Solution:

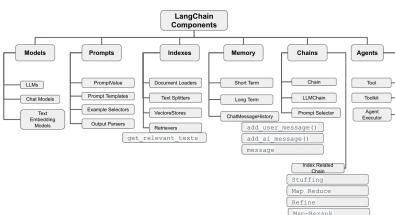
- Memory management built-in
- Easy data integration
- Tool calling framework
- Simple, modular code



(Ref: <https://www.leanware.co/insights/langchain-vs-pinecone-which-should-you-choose>)

Bottom Line: Build AI apps faster with less code

Core Building Blocks



Foundation:

- **Models:** Connect to any LLM
- **Prompts:** Dynamic templates
- **Memory:** Conversation context

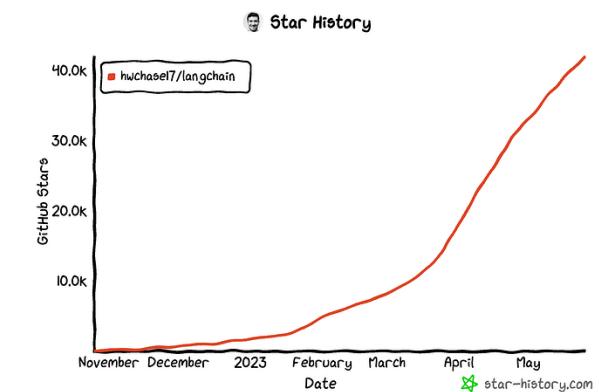
Advanced:

- **Retrievers:** Search your data
- **Agents:** Autonomous reasoning
- **LCEL:** Chain components easily

Mix and match components to build your app

So, Why LangChain?

- **RAG Applications:** Connect LLMs to your data
- **Intelligent Agents:** Autonomous tool usage
- **Production Ready:** Monitoring & logging built-in
- **Model Flexibility:** Switch providers easily
- **Modular Design:** Use only what you need



(Ref: LangChain tutorial: Build an LLM-powered app)

The LangChain Ecosystem

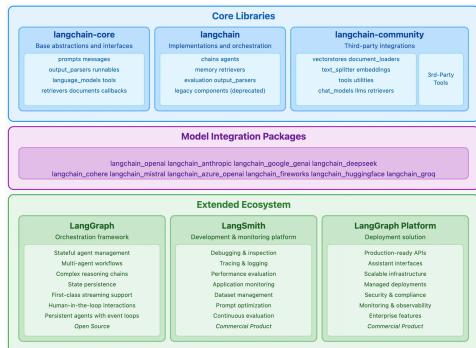
Community:

- 80,000+ GitHub stars
- 2,000+ contributors
- Millions of downloads/month
- Active Discord community

License:

- MIT - Commercial friendly

Ecosystem Tools:



(Ref: Demystifying the LangChain Ecosystem for LLM-Powered Application Development - Wltanksalpa)

- **LangGraph:** Complex agents
- **LangServe:** Deploy APIs
- **LangSmith:** Monitor & debug

(Ref: What is Langchain and why should I care as a developer? - Logan Kilpatrick)

Installation & Setup

Python 3.10 or higher

```
# Core packages
pip install langchain langchain-core

# Provider-specific packages for Groq
pip install langchain-groq

# Alternative providers
# pip install langchain-anthropic langchain-google-genai

# For document processing
pip install langchain-community langchain-text-splitters

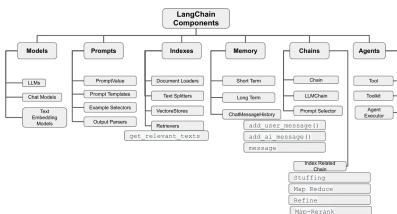
# For embeddings
pip install langchain-huggingface

# For legacy features (if needed)
pip install langchain-classic

//API Keys Setup:
import os
os.environ["GROQ_API_KEY"] = "your-groq-api-key-here"
# Or use .env file with python-dotenv
```

```
# Optional: Enable LangSmith tracing
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = "your-langsmith-key"
```

Core Components Overview



| Component | Purpose | Use Case |
|------------|---------------------|-----------------------|
| Models | Connect to LLMs | Text generation, chat |
| Prompts | Template management | Dynamic prompts |
| LCEL | Chain components | Build workflows |
| Memory | Store conversations | Chatbots |
| Retrievers | Search data | RAG, Q&A |
| Agents | Tool selection | Autonomous tasks |

(Ref: LangChain Use Cases Documentation)

Implementation

Implementations

Use as LLM

Sentiment Analysis: Zero Shot

```
from langchain_groq import ChatGroq
from langchain.core.prompts import ChatPromptTemplate
from langchain.core.output_parsers import StrOutputParser

# 1. Define the LLM
model = ChatGroq(model_name="gemma2-9b-it", temperature=0)

# 2. Create a prompt template
prompt = ChatPromptTemplate.from_template(
    "Analyze the sentiment of the following text.\n"
    "Respond with only one word: Positive, Negative, or Neutral.\n"
    "Text: {input_text}"
)

# 3. Create a simple output parser
parser = StrOutputParser()

# 4. Build the LCEL chain
chain = prompt | model | parser

# Define input text
input_text = "I love LangChain! It's the best NLP library I've ever used."

# 5. Invoke the chain
sentiment = chain.invoke({"input_text": input_text})

# Print the sentiment
print(sentiment)
```

(Ref: How LangChain Makes Large Language Models More Powerful)

Key Concepts: Chains vs Agents

- **Chains (LCEL):**
 - Predetermined sequence of operations
 - Composed with pipe operator: `prompt | llm | parser`
 - Fixed execution path
 - Best for: Structured, predictable workflows

- **Agents:**
 - Dynamic decision-making with LLM reasoning
 - Choose tools based on input
 - Adaptive execution path
 - Best for: Complex, unpredictable scenarios

When to use what?

- Use **Chains/LCEL** when: Steps are known, workflow is fixed
- Use **Agents** when: Need dynamic tool selection, multi-step reasoning

(Ref: Superpower LLMs with Conversational Agents)

What Can You Build?

RAG Applications:

- Document Q&A
- Knowledge base search
- Semantic search

Conversational AI:

- Chatbots with memory
- Customer support
- Personal assistants

Data Analysis:

- SQL query generation
- Report generation
- Data insights

Autonomous Agents:

- Web research
- API integration
- Multi-step workflows

Named Entity Recognition (NER): Zero Shot

```
from langchain_groq import ChatGroq
from langchain_core.prompts import
    ChatPromptTemplate
from langchain_core.output_parsers import
    StrOutputParser

# 1. Setup Model
model = ChatGroq(model="gemma2-9b-it")

# 2. Create Prompt for NER extraction
template = """
Identify the named entities (Person, Organization,
Amount) in the text.
Format the output as a bulleted list.

Text: {text}
"""
prompt = ChatPromptTemplate.from_template(template)

# 3. Build Chain
chain = prompt | model | StrOutputParser()

# 4. Invoke
input_text = "Microsoft is acquiring Nuance
    Communications for $19.7 billion."
entities = chain.invoke({"text": input_text})
print(entities)
```

Use as Chains

LCEL: Basic Example

```
// Old Pattern (Deprecated):
from langchain.llms import OpenAI
from langchain.chains import LLMChain

llm = OpenAI()
chain = LLMChain(llm=llm, prompt=prompt)
result = chain.run("input")

// Modern LCEL Pattern with Groq:
from langchain_groq import ChatGroq
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Set up Groq model, e.g., Gemma or Llama 3
llm = ChatGroq(model_name="gemma-7b-it")
prompt = ChatPromptTemplate.from_template("Tell me about
    {topic}")
output_parser = StrOutputParser()

# Build chain with pipe operator
chain = prompt | llm | output_parser

# Invoke the chain
result = chain.invoke({"topic": "LangChain"})
```

LCEL: Advanced Features

```
// Streaming Support:
chain = prompt | llm | output_parser

# Stream tokens as they're generated
for chunk in chain.stream({"topic": "AI"}):
    print(chunk, end="", flush=True)

// Async Execution:
# Async invocation
result = await chain.invoke({"topic": "AI"})

# Async streaming
async for chunk in chain.astream({"topic": "AI"}):
    print(chunk, end="", flush=True)

// Batch Processing:
# Process multiple inputs in parallel
results = chain.batch([
    {"topic": "AI"},
    {"topic": "ML"},
    {"topic": "LangChain"}
])
```

LCEL: Complex Chains

```
// Multi-step Chain with RunnablePassthrough:
from langchain_core.runnables import RunnablePassthrough

chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | llm
    | output_parser
)

result = chain.invoke("What is LangChain?")

// Parallel Execution with RunnableParallel:
from langchain_core.runnables import RunnableParallel

chain = RunnableParallel(
    summary=prompt1 | llm | output_parser,
    keywords=prompt2 | llm | output_parser
)

result = chain.invoke({"text": "Long document..."})
# Returns: {"summary": "...", "keywords": "..."}
```

Model Integration: Modern Approach

```
Updated Import Structure:

# OpenAI models
from langchain_openai import ChatOpenAI, OpenAIEMBEDDINGS

# Hugging Face models
from langchain_huggingface import HuggingFaceEndpoint

# Google models (Gemini)
from langchain_google_genai import ChatGoogleGenerativeAI

# Anthropic models
from langchain_anthropic import ChatAnthropic

// Example Usage:
from langchain_groq import ChatGroq
from langchain_core.messages import HumanMessage, SystemMessage

# Initialize Groq with Gemma
chat = ChatGroq(model="gemma-2-9b-it", temperature=0.7)

messages = [
    SystemMessage(content="You are a helpful assistant"),
    HumanMessage(content="Explain LangChain in 2 sentences")]
```

```
] response = chat.invoke(messages)
print(response.content)
```

Use as RAG

Complete RAG Application: A Quick Start Example

```
from langchain_groq import ChatGroq
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.document_loaders import WebBaseLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

# Load and process documents
loader = WebBaseLoader("https://example.com/doc")
docs = loader.load()
splitter = RecursiveCharacterTextSplitter(chunk_size=1000)
splits = splitter.split_documents(docs)

embeddings = HuggingFaceEmbeddings()
vectorstore = Chroma.from_documents(splits, embeddings)
retriever = vectorstore.as_retriever()

prompt = ChatPromptTemplate.from_template("""
Answer based on context: {context}
Question: {question}""")

model = ChatGroq(model_name="llama3-8b-8192")

# RunnablePassthrough => "context": retriever("What is the main
topic?"),
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt | model | StrOutputParser()

response = chain.invoke("What is the main topic?")
```

Use as Agent

Example: Modern Agent in LangChain v1

- No more AgentExecutor or create_tool_calling_agent
- Built on LangGraph internally
- Returns full message history

```
# pip install -qU langchain langchain_groq
from langchain_agents import create_agent
from langchain_core.tools import tool

@tool
def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

# Create agent (model can be string or ChatGroq object)
agent = create_agent(
    model="llama-3-3-70b-versatile",
    tools=[get_weather],
    system_prompt="You are a helpful assistant."
)

# Run the agent
response = agent.invoke({})
```

```

    "messages": [{"role": "user", "content": "What is the weather in
    Pune?"}]}
}

# Extract final response
final_message = response["messages"][-1]
print(final_message.content)

```

Framework

LangChain Framework Components

Framework Architecture

Chains: The core of LangChain. Components (and even other chains) can be stringed together to create *chains*.
Prompt templates: Prompt templates are templates for different types of prompts. Like "chatbot" style templates, ELI5 question-answering, etc
LLMs: Large language models like GPT-3, BLOOM, etc
Indexing Utils: Ways to interact with specific data (embeddings, vectorstores, document loaders)
Tools: Ways to interact with the outside world (search, calculators, etc)
Agents: Agents use LLMs to decide what actions should be taken. Tools like web search or calculators can be used, and all are packaged into a logical loop of operations.
Memory: Short-term memory, long-term memory.

Core Building Blocks:

- **Models:** LLMs, Chat Models, Embeddings
- **Prompts:** Dynamic template management
- **Output Parsers:** Structured output extraction
- **Retrievers:** Document and data access
- **Memory:** Conversation state persistence
- **Agents & Tools:** Dynamic reasoning and actions

(Ref: Building the Future with LLMs, LangChain, & Pinecone)

Models

Models in LangChain

Three Types of Models:

- **LLMs (Large Language Models):**
 - Input: String (prompt)
 - Output: String (completion)
 - Examples: GPT-4, Claude, Gemma, Llama 3, Mixtral.
 - Use case: Text generation, completion
- **Chat Models:**
 - Input: List of messages
 - Output: Chat message
 - Examples: ChatGPT, Claude Chat, ChatGroq

- Use case: Conversational AI

- **Embedding Models:**

- Input: Text
- Output: Vector (list of floats)
- Examples: OpenAI Embeddings, HuggingFace, Sentence Transformers
- Use case: Semantic search, similarity

Models Functionality

- Tool calling: calling external tools (like databases queries or API calls) and use results in their responses.
- Structured output: where the model's response is constrained to follow a defined format.
- Multimodality: process and return data other than text, such as images, audio, and video.
- Reasoning: models perform multi-step reasoning to arrive at a conclusion.

Models Usage

- With agents: Models can be dynamically specified when creating an agent.
- Standalone: Models can be called directly (outside of the agent loop) for tasks like text generation, classification, or extraction without the need for an agent framework.

Model Integration: Modern Syntax

For standalone model `init_chat_model`

```

from langchain.chat_models import init_chat_model # Still valid in v1
# OR use provider-specific import (recommended)
from langchain.groq import ChatGroq
from langchain.messages import HumanMessage, AIMessage,
    SystemMessage

# Initialize Groq LLM (ensure GROQ_API_KEY is set in your
# environment)
model = ChatGroq(model="llama-3.3-70b-versatile") #
    https://console.groq.com/docs/models

# # or assuming os.environ["ANTHROPIC_API_KEY"] = "sk-..."
# model = init_chat_model(
#     "claude-sonnet-4-5-20250929",
#     # Kwargs passed to the model:
#     temperature=0.7,
#     timeout=30,
#     max_tokens=1000,
# )

conversation = [
    SystemMessage("You are a helpful assistant that translates English
        to French."),
    HumanMessage("Translate: I love programming."),
    AIMessage("J'adore la programmation."),
    HumanMessage("Translate: I love building applications.")
]

response = model.invoke(conversation)
print(response) # AIMessage("J'adore creer des applications.")

```

Multimodal

Certain models can process and return non-textual data such as images, audio, and video. You can pass non-textual data to a model by providing content blocks.

```

response = model.invoke("Create a picture of a cat")
print(response.content_blocks)
# [
#     {"type": "text", "text": "Here's a picture of a cat"},
#     {"type": "image", "base64": "...", "mime_type": "image/jpeg"},
# ]

```

Reasoning

Many models are capable of performing multi-step reasoning to arrive at a conclusion.

```

## Streaming
for chunk in model.stream("Why do parrots have colorful feathers?"):
    reasoning_steps = [r for r in chunk.content_blocks if r["type"] ==
        "reasoning"]
    print(reasoning_steps if reasoning_steps else chunk.text)

## Complete Output
response = model.invoke("Why do parrots have colorful feathers?")
reasoning_steps = [b for b in response.content_blocks if b["type"] ==
    "reasoning"]
print(" ".join(step["reasoning"] for step in reasoning_steps))

```

Prompts

Prompts in LangChain

Prompt Management Strategies:

- **Prompt Templates:**
 - Parameterized templates with variables
 - Dynamic input insertion
 - Reusable prompt structures
- **Chat Prompt Templates:**
 - Multi-message conversations
 - System, human, AI message roles
 - Better for chat models
- **Few-Shot Prompts:**
 - Include example inputs/outputs
 - Guide model response style
 - Improve accuracy

Prompt Templates: Modern Examples

```
from langchain.core.prompts import PromptTemplate

# Use in chain with Groq
from langchain_groq import ChatGroq
from langchain_core.output_parsers import StrOutputParser

template = """You are a {role} assistant.
Task: {task}
Context: {context}
Provide a {format} response."""

prompt = PromptTemplate(
    template=template,
    input_variables=['role', 'task', 'context', 'format']
)

formatted = prompt.format(
    role="helpful",
    task="explain quantum computing",
    context="for beginners",
    format="simple"
)

chain = prompt | ChatGroq(model="llama-3.3-70b-versatile") |
    StrOutputParser()
result = chain.invoke({
    "role": "helpful",
    "task": "explain quantum computing",
    "context": "for beginners",
    "format": "simple"
})

print(result)
```

Chat Prompting

```
# Chat Prompt Template:
from langchain.core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are {role}"),
    ("human", "{input}"),
    ("ai", "I understand. Let me help with that."),
    ("human", "{follow_up}")
])

chain = prompt | ChatGroq(model="llama-3.3-70b-versatile") |
    StrOutputParser()
result = chain.invoke({
    "role": "helpful",
    "input": "Explain quantum computing",
    "follow_up": "Make it simpler"
})
print(result)
```

Document Loaders & Retrievers

Document Loading

Wide Range of Loaders:

- **Files:** PDF, Word, PowerPoint, CSV, Markdown
- **Web:** HTML, URLs, sitemaps, web scraping
- **Cloud:** S3, GCS, Google Drive, Notion
- **Databases:** SQL, MongoDB, Elasticsearch
- **Communication:** Email, Slack, Discord
- **Code:** GitHub, GitLab, Jupyter notebooks

Modern Document Processing:

```
from langchain_community.document_loaders import (
    PyPDFLoader,
    WebBaseLoader,
    TextLoader)
from langchain_text_splitters import RecursiveCharacterTextSplitter

loader = PyPDFLoader("document.pdf")
documents = loader.load()

splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200)

chunks = splitter.split_documents(documents)
```

Vector Stores & Retrievers

Popular Vector Stores:

- **Chroma:** Open-source, local-first, easy setup
- **Pinecone:** Managed service, production-ready
- **Weaviate:** GraphQL API, hybrid search
- **Qdrant:** High performance, filtering support
- **LanceDB:** Serverless, embedded option

Complete Example with Chroma & HuggingFace Embeddings:

```
# from langchain_community.embeddings import
# HuggingFaceEmbeddings
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import Chroma
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader

loader = WebBaseLoader("https://example.com/article")
docs = loader.load()

splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
    chunk_overlap=200)
splits = splitter.split_documents(docs)

vectorstore = Chroma.from_documents(
    documents=splits,
    embedding=HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2"))

retriever = vectorstore.as_retriever(search_kwargs={"k": 3})
```

Retrieval Strategies

Different Retrieval Methods:

```
# Returns the most semantically similar documents to the user
# query based purely on vector distance (top-k closest matches).
retriever = vectorstore.as_retriever(search_type="similarity",
    search_kwargs={"k": 4})

# MMR (Maximum Marginal Relevance) Balances relevance and
# diversity,
# ensuring the retrieved documents aren't repetitive by reducing
# semantic redundancy among the top-k results.
retriever = vectorstore.as_retriever(search_type="mmr",
    search_kwargs={"k": 4, "fetch_k": 20})

# Filters out documents that fall below a minimum similarity score,
# ensuring only high-confidence matches are returned (still limited
# by k)
retriever = vectorstore.as_retriever(search_type =
    "similarity_score_threshold",
    search_kwargs={"score_threshold": 0.8, "k": 4})
```

```
# Manual search: Directly retrieves the top-k most similar documents
# without using a retriever wrapper same as search_type "similarity"
# but explicitly called.
results = vectorstore.similarity_search("What is machine learning?", k=3)

# With scores: Same as above but also returns the similarity
# score for each document, helping you inspect retrieval quality
results_with_scores = vectorstore.similarity_search_with_score(
    "What is machine learning?", k=3)
```

Chains with LCEL

Modern LangChain: LCEL (LangChain Expression Language)

What is LCEL?

- Modern, declarative way to build chains (introduced 2023)
- Uses pipe operator | to chain components
- Replaces old LLMChain pattern
- Built-in streaming, async, and batch support

Key Benefits:

- **Simplicity:** More readable and concise
- **Streaming:** Built-in streaming by default
- **Async:** Native async/await support
- **Observability:** Better tracing and debugging
- **Fallbacks:** Easy error handling and retries

(Ref: LangChain LCEL Documentation)

Modern Chains: LCEL Overview

What Changed?

- **Old:** LLMChain, SimpleSequentialChain (Deprecated)
- **New:** LCEL with pipe operator |

Core Runnables:

- **RunnablePassthrough:** Pass data through
- **RunnableParallel:** Execute in parallel
- **RunnableLambda:** Custom functions
- **RunnableBranch:** Conditional execution

LCEL: Basic Chain Patterns

```
// Simple Chain:
from langchain_groq import ChatGroq
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

prompt = ChatPromptTemplate.from_template("Tell me a joke about {topic}")
model = ChatGroq(model_name="gemma-7b-it")
output_parser = StrOutputParser()

chain = prompt | model | output_parser
result = chain.invoke({"topic": "programming"})

// Chain with Multiple Steps:
# Step 1: Generate topic
topic_chain = (
    ChatPromptTemplate.from_template("Suggest a {genre} topic")
    | ChatGroq(model_name="gemma-7b-it")
    | StrOutputParser()
)

# Step 2: Write content
content_chain = (
    ChatPromptTemplate.from_template("Write a story about {topic}")
    | ChatGroq(model_name="llama3-8b-8192")
    | StrOutputParser()
)

# Combine: topic generates input for content
full_chain = {"topic": topic_chain} | content_chain
result = full_chain.invoke({"genre": "science fiction"})
```

LCEL: RAG Chain Pattern

Retrieval Augmented Generation:

```
from langchain_core.runnables import RunnablePassthrough
from langchain_groq import ChatGroq
# Assume 'retriever' from previous slide is defined

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

# RAG chain
rag_chain = (
    {
        "context": retriever | format_docs,
        "question": RunnablePassthrough()
    }
    | ChatPromptTemplate.from_template("""
        Answer the question based on the context:
        Context: {context}
        Question: {question}
    """)
    | ChatGroq(model_name="llama3-8b-8192")
    | StrOutputParser()
)

# Use it
answer = rag_chain.invoke("What is LangChain?")
```

LCEL: Parallel Execution

Benefits:

- Faster execution
- Clean code structure
- Easy to add/remove tasks

Run Multiple Chains in Parallel:

```
from langchain_core.runnables import RunnableParallel
from langchain_groq import ChatGroq
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Define parallel tasks
parallel_chain = RunnableParallel(
    summary=ChatPromptTemplate.from_template("Summarize: {text}")
    | ChatGroq(model_name="llama3-8b-8192")
    | StrOutputParser(),

    keywords=ChatPromptTemplate.from_template("Extract keywords: {text}")
    | ChatGroq(model_name="gemma-7b-it")
    | StrOutputParser(),

    sentiment=ChatPromptTemplate.from_template("Analyze sentiment: {text}")
    | ChatGroq(model_name="gemma-7b-it")
    | StrOutputParser()
)

# Execute all in parallel
results = parallel_chain.invoke({"text": "Long document content..."})
```

LCEL: Error Handling & Fallbacks

```
// Fallback to Alternative Model:
from langchain_groq import ChatGroq
from langchain_anthropic import ChatAnthropic

primary = ChatGroq(model_name="llama3-70b-8192")
fallback = ChatAnthropic(model="claude-3-opus-20240229")

chain = prompt | primary.with_fallbacks([fallback]) | output_parser

// Retry Logic: If Groq fails, automatically retries
from langchain_core.runnables import RunnableRetry

chain_with_retry = (
    prompt
    | RunnableRetry(max_attempts=3, wait_exponential_jitter=True)
    | ChatGroq(model_name="gemma-7b-it")
    | output_parser
)

// Custom Error Handling:
from langchain_core.runnables import RunnableLambda

def handle_errors(x):
    try:
        return x
    except Exception as e:
        return {"error": str(e)}

chain = prompt | model | RunnableLambda(handle_errors)
```

LCEL: Streaming

```
// Stream Tokens as Generated:
from langchain_groq import ChatGroq
# Assume prompt and output_parser are defined
chain = prompt | ChatGroq(model_name="gemma-7b-it") | StrOutputParser()

# Stream output
for chunk in chain.stream({"topic": "AI"}):
    print(chunk, end="", flush=True)

// Async Streaming:
import asyncio

async def stream_response():
    async for chunk in chain.astream({"topic": "AI"}):
        print(chunk, end="", flush=True)

    asyncio.run(stream_response())
```

```
// Streaming with Events:
# Get detailed streaming events
async for event in chain.astream_events({"topic": "AI"}, version="v1"):
    kind = event["event"]
    if kind == "on_chat_model_stream":
        content = event["data"]["chunk"].content
        print(content, end="", flush=True)
```

Memory

Memory in LangChain

Why Memory?

- LLMs are stateless by default
- Chatbots need conversation context
- Memory stores and retrieves conversation history

Memory Types (Consolidated):

- ConversationBufferMemory:
 - Stores entire conversation history
 - Simple but can exceed token limits
 - Best for: Short conversations
- ConversationBufferWindowMemory:
 - Keeps only last K interactions
 - Prevents token overflow
 - Best for: Longer conversations with recent context
- ConversationSummaryMemory:
 - Summarizes old messages
 - Reduces token usage
 - Best for: Very long conversations

Memory: Implementation Examples

Important Note

In LangChain v1, memory classes have moved to langchain-classic. To use pip install langchain-classic

```
from langchain_classic.memory import ConversationBufferMemory
from langchain_groq import ChatGroq
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.runnables import RunnablePassthrough

memory = ConversationBufferMemory(
    return_messages=True,
    memory_key="history")

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant"),
    MessagesPlaceholder(variable_name="history"),
    ("human", {"input"}))

chain = (
```

```

RunnablePassthrough.assign(history=lambda x:
    memory.load_memory_variables({})["history"]  )
| prompt
| ChatGroq(model_name="gemma2-9b-it"))

# Use the chain
response = chain.invoke({"input": "Hi, I'm Alice"})
memory.save_context({"input": "Hi, I'm Alice"}, {"output": response.content})

```

Memory: Window Memory

Benefits:

- Fixed memory footprint
- Maintains recent context
- Prevents token limit issues

Buffer Window Memory (Keeps Last K):

```

from langchain.memory import ConversationBufferWindowMemory
memory = ConversationBufferWindowMemory(
    k=3, # Only keeps last 3 interactions
    return_messages=True,
    memory_key="history"
)

conversations = [ # Add conversations
    ("Tell me about Python", "Python is a versatile language..."),
    ("What makes it popular?", "Its simplicity and libraries..."),
    ("Give an example", "Like NumPy for computing..."),
    ("What about AI?", "Python excels in AI with TensorFlow...")
]

for input_text, output_text in conversations:
    memory.save_context({"input": input_text}, {"output": output_text})

# Retrieves only last 3 interactions
history = memory.load_memory_variables({})
print(f"Stored messages: {len(history['history'])}") # Will be 6 (3 pairs)

```

Agents & Tools

Modern Agents Overview

What Are Agents?

- Use LLMs as reasoning engines
- Dynamically choose which tools to use
- Iterative: observe, think, act, repeat
- Best for complex, multi-step tasks

Modern Agent Types (LangChain v1):

- **create_agent**: Primary method, uses ReAct pattern
- **create_deep_agent**: For complex, long-running tasks

When to Use:

- **create_agent**: Simple chatbots, Q&A, tool calling
- **create_deep_agent**: Research, multi-step planning, file-system access
- **LangGraph**: Custom workflows requiring fine-grained control

(Ref: LangChain Agents Documentation)

Creating Tools: Modern Approach

Tool Requirements:

- Must have a docstring (used by LLM to understand tool purpose)
- Type hints are required for parameters
- Return type should be string or JSON-serializable

```

// Method 1: Using @tool Decorator:
from langchain.core.tools import tool

@tool
def search_wikipedia(query: str) -> str:
    """Search Wikipedia for information about a topic."""
    from wikipedia import summary
    try:
        return summary(query, sentences=3)
    except:
        return "Could not find information."

@tool
def calculate(expression: str) -> str:
    """Evaluate a mathematical expression."""
    try:
        return str(eval(expression))
    except:
        return "Invalid expression"

```

Creating Tools: Modern Approach

// Method 2: From Function:

```

from langchain.core.tools import Tool

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

weather_tool = Tool.from_function(
    func=get_weather,
    name="weather",
    description="Get current weather for a location")

```

Modern Agent: create_agent()

- Built on LangGraph (durable execution)
- Automatic tool calling loop
- Simple, unified API

```

from langchain.agents import create_agent
from langchain.core.tools import tool
from langchain.groq import ChatGroq

@tool
def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"It's sunny in {city}!"

@tool
def search_wikipedia(query: str) -> str:
    """Search Wikipedia for information."""
    return f"Wikiresults for: {query}"

# Create agent with tools
agent = create_agent(
    model="llama-3.3-70b-versatile", # Can use model string or
                                    # ChatGroq object
    tools=[get_weather, search_wikipedia],
    system_prompt="You are a helpful assistant that can check weather
and search Wikipedia.")

```

```

# Run the agent
response = agent.invoke({"messages": [{"role": "user", "content": "What's the weather in Paris?"}]})
print(response["messages"][-1].content)

```

Modern Approach with bind_tools

```

from langchain.groq import ChatGroq
from langchain.core.tools import tool

@tool
def multiply(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b

@tool
def add(a: int, b: int) -> int:
    """Add two numbers."""
    return a + b

# Bind tools to model
llm = ChatGroq(model_name="llama3-8b-8192")
llm.with_tools = llm.bind_tools([multiply, add])

# Invoke
response = llm.with_tools.invoke("What is 3 times 4 plus 5?")

# Check if tool was called
if response.tool_calls:
    tool_call = response.tool_calls[0]
    print(f"Tool: {tool_call['name']}") # prints "Tool: multiply"
    print(f"Args: {tool_call['args']}") # prints "Args: [3, 4]"

```

Agent Middleware

Extend Agent Behavior Without Modifying Core Logic:

- **Human-in-the-loop**: Approve tool calls before execution
- **Conversation compression**: Summarize long conversations
- **PII redaction**: Remove sensitive data
- **Rate limiting**: Control API usage
- **Error handling**: Retry failed operations

```

from langchain.agents import create_agent
from langchain.agents.middleware import AgentMiddleware

# Example: Logging middleware
class LoggingMiddleware(AgentMiddleware):
    async def on_agent_start(self, state, context):
        print(f"Agent started with input:
{state['messages'][-1].content}")

    async def on_agent_end(self, state, context):
        print(f"Agent completed with output:
{state['messages'][-1].content}")

# Create agent with middleware
agent = create_agent(
    model="llama-3.3-70b-versatile",
    tools=[get_weather],
    middleware=[LoggingMiddleware()])

```

Output Parsers

Output Parsers Overview

Why Output Parsers?

- LLMs return unstructured text
- Applications need structured data
- Parsers extract and validate output

Common Parser Types:

- **StrOutputParser**: Basic string extraction
- **JsonOutputParser**: Parse JSON responses
- **PydanticOutputParser**: Structured data with validation
- **StructuredOutputParser**: Multiple fields
- **CommaSeparatedListOutputParser**: Lists

Modern Best Practice:

- Use OpenAI's `with_structured_output()` when possible
- More reliable than prompt-based parsing
- Leverages function calling internally

Structured Output: Modern Approach

Benefits:

- Type-safe responses
- Automatic validation
- More reliable than prompt-based parsing

```
from langchain_groq import ChatGroq
from pydantic import BaseModel, Field

# Define output schema
class Person(BaseModel):
    name: str = Field(description="Person's name")
    age: int = Field(description="Person's age")
    occupation: str = Field(description="Person's job")

# Create model with structured output (relies on tool-calling)
llm = ChatGroq(model_name="llama3-70b-8192")
structured_llm = llm.with_structured_output(Person)

# Use in chain
response = structured_llm.invoke(
    "Tell me about a software engineer named Alice who is 28"
)

# Response is a Pydantic object
print(response.name)
print(response.age)
print(response.occupation)
```

Pydantic Output Parser (Alternative)

```
from langchain.output_parsers import PydanticOutputParser
from langchain_groq import ChatGroq
from pydantic import BaseModel, Field, validator
from typing import List

class MovieReview(BaseModel):
    title: str = Field(description="Movie title")
    rating: int = Field(description="Rating from 1-10")

parser = PydanticOutputParser(pydantic_object=MovieReview)

# Add to prompt
prompt = ChatPromptTemplate.from_template("""
Review the movie: {movie}

{format_instructions}""")

chain =
    (prompt.partial(format_instructions=parser.get_format_instructions())
     | ChatGroq(model_name="gemma-7b-it") | parser)

result = chain.invoke({"movie": "Inception"})
```

Output Parser Error Handling

```
from langchain_groq import ChatGroq
# Assume 'parser' is a defined PydanticOutputParser

// OutputFixingParser (Auto-fix Errors):
from langchain.output_parsers import OutputFixingParser

# If parsing fails, use LLM to fix
fixing_parser = OutputFixingParser.from_llm(
    parser=parser,
    llm=ChatGroq(model_name="gemma-7b-it")
)

// Automatically fixes malformed output
# result = fixing_parser.parse(malformed_output)

// RetryOutputParser (Retry with Context):
from langchain.output_parsers import RetryWithErrorOutputParser

retry_parser = RetryWithErrorOutputParser.from_llm(
    parser=parser,
    llm=ChatGroq(model_name="llama3-8b-8192")
)

# Retries with both output and original prompt
# result = retry_parser.parse_with_prompt(...)
```

LangChain Ecosystem

LangChain Ecosystem Components

- **LangChain Core**:
 - Base abstractions and LCEL
 - Foundation for all other packages
- **LangChain Community**:
 - Third-party integrations
 - Vector stores, document loaders
 - Community-maintained tools
- **LangGraph**:

- Build stateful, multi-actor applications
- Complex agent workflows with cycles
- State management for agents

• LangServe:

- Deploy chains as REST APIs
- Automatic FastAPI generation
- Production deployment

• LangSmith:

- Debugging and monitoring
- Tracing and evaluation
- Dataset management

LangGraph: Stateful Agents

- Build complex, stateful agent workflows
- Support for cycles and conditional logic
- Persist state across interactions
- Multiple agents working together

```
from langgraph.graph import StateGraph
from typing import TypedDict, Annotated
import operator

class AgentState(TypedDict):
    messages: Annotated[List, operator.add]
    next: str

def call_model(state):
    response = llm.invoke(state["messages"])
    return {"messages": [response]}

def should_continue(state):
    if len(state["messages"]) > 5:
        return "end"
    return "continue"

workflow = StateGraph(AgentState)
workflow.add_node("agent", call_model)
workflow.add_conditional_edges("agent", should_continue)
workflow.set_entry_point("agent")

app = workflow.compile()
```

LangServe: Deploy as API

Automatic Features:

- Interactive playground at `/chat/playground`
- OpenAPI docs at `/docs`
- Streaming support
- Batch processing

Deploy Any Chain as REST API:

```

from fastapi import FastAPI
from langserve import add_routes
from langchain_groq import ChatGroq
from langchain.core.prompts import ChatPromptTemplate

# Create chain
prompt = ChatPromptTemplate.from_template("Tell me about
    {topic}")
chain = prompt | ChatGroq(model_name="gemma-7b-it")

# Create FastAPI app
app = FastAPI(
    title="LangChain API", version="1.0",
    description="API for my LangChain application")

# Add chain as route
add_routes(app, chain, path="/chat")

# Run: uvicorn app:app --reload
# API available at: http://localhost:8000/chat

```

LangSmith: Monitoring & Debugging

What is LangSmith?

- Platform for debugging LLM applications
- Trace every step of chain execution
- Evaluate model performance
- Manage test datasets
- Monitor production applications

Setup:

```

import os
from langchain_groq import ChatGroq
from langchain.core.prompts import ChatPromptTemplate
from langchain.core.output_parsers import StrOutputParser

os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = "your-langsmith-api-key"
os.environ["LANGCHAIN_PROJECT"] = "my-groq-project"

# Now all chain executions are automatically traced
prompt = ChatPromptTemplate.from_template("Tell me about
    {topic}")
llm = ChatGroq(model_name="gemma-7b-it")
output_parser = StrOutputParser()
chain = prompt | llm | output_parser
result = chain.invoke({"topic": "Large Language Models"})

# View traces at: https://smith.langchain.com

```

Best Practices

Best Practices: Error Handling

```

// 1. Use Fallbacks:
from langchain_groq import ChatGroq
from langchain_anthropic import ChatAnthropic

primary = ChatGroq(model_name="llama3-70b-8192")
fallback = ChatAnthropic(model="claude-3-opus-20240229")

chain = prompt | primary.with_fallbacks([fallback]) | parser

// 2. Implement Retries:
from langchain_core.runnables import RunnableRetry

chain_with_retry = (

```

```

prompt
| RunnableRetry(
    max_attempts=3,
    wait_exponential_jitter=True
)
| ChatGroq(model_name="gemma-7b-it")
| parser
)

// 3. Graceful Degradation:
try:
    result = chain.invoke(input_data)
except Exception as e:
    # logger.error(f"Chain failed: {e}")
    # result = fallback_response()
    pass

```

Best Practices: Token Management with Groq

Groq Models - Fast and Free Tier:

- gemma2-9b-it: Ultra-fast, 8K context, great for chat
- llama3-8b-8192: Balanced, 8K context, general purpose
- llama3-70b-8192: Most capable, 8K context, complex tasks
- llama-3.1-8b-instant: Fastest, optimized for low latency
- Groq billed by requests/day on free tier, not tokens

```

from langchain_groq import ChatGroq

# Choose model based on needs
llm_fast = ChatGroq(
    model_name="gemma2-9b-it",
    max_tokens=500,
    temperature=0.7,
    max_retries=2
)

# Monitor context length manually
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("google/gemma-2-9b-it")
token_count = len(tokenizer.encode(text))

# Groq's speed allows batch processing without latency concerns

```

Best Practices: Security & Privacy

- Be aware of the data usage policies of your LLM provider.
- Groq has a zero-retention policy for API data.
- Consider self-hosted models for maximum data control.
- Implement PII detection and redaction before sending data.

```

// 1. API Key Management:
import os
from dotenv import load_dotenv

load_dotenv() # Load from .env file
api_key = os.getenv("GROQ_API_KEY")

// 2. Timeouts and Retries:
from langchain_groq import ChatGroq

```

```

llm = ChatGroq(model_name="llama3-8b-8192",
    temperature=0,
    max_retries=2,
    request_timeout=30) # seconds

// 3. Input Validation:
def validate_input(user_input: str) -> str:
    if len(user_input) > 8000: # Sanitize input
        raise ValueError("Input too long for the model context.")
    # Remove potential injection attempts
    return user_input.strip()

```

Best Practices: Choosing the Right Model

Groq Model Selection Guide:

| Use Case | Model | Why |
|-------------------|----------------------|-------------------------|
| Chatbots | gemma2-9b-it | Fast, efficient, good r |
| Summarization | llama3-8b-8192 | Balanced speed/qual |
| Complex reasoning | llama3-70b-8192 | Most capable |
| Ultra-low latency | llama-3.1-8b-instant | Optimized for speed |
| Code generation | llama3-70b-8192 | Better instruction fol |

```

# Pattern: Start with fast model, fallback to capable model
from langchain_groq import ChatGroq

primary = ChatGroq(model_name="gemma2-9b-it",
    temperature=0.7)
fallback = ChatGroq(model_name="llama3-70b-8192",
    temperature=0.7)

chain = prompt | primary.with_fallbacks([fallback]) | parser

```

Best Practices: Async Patterns

Benefits:

- Faster parallel processing
- Better resource utilization
- Improved user experience with streaming

```

// Use Async for Better Performance:
import asyncio

async def process_multiple_queries(queries):
    # Process queries concurrently
    tasks = [chain.invoke({"input": q}) for q in queries]
    results = await asyncio.gather(*tasks)
    return results

# Run
queries = ["Query 1", "Query 2", "Query 3"]
results = asyncio.run(process_multiple_queries(queries))
// Async Streaming:
async def stream_response(input_text):
    async for chunk in chain.astream({"input": input_text}):
        print(chunk, end="", flush=True)

asyncio.run(stream_response("Tell me about AI"))

```

What's New

What's New in LangChain Ecosystem

(Oct 2025, Release of 1.0 version)

Major Additions in LangChain/Graph 1.0

- LangChain is now on top of LangGraph (not the other way round, as before)
- LangChain agent `create_agent` (ReACT) is one specialized case of many types in LangGraph. You can even visualize it as a workflow of nodes.
- Middle-ware to control the data before going to LLM. Curation, summarization, context engineering, etc.
- Standard content blocks to cater to different output formats by different LLMs.

Code Reorganization in LangChain/Graph 1.0

This is in line with principle of 'Simplified Name-spaces'.

- LangChain legacy chains, hub, vector stores, retrievers etc are now part of `langchain_classic` module.
- All agents are in `langchain.agents`
- All Models are in `langchain.chat_models`
- All Tools are in `langchain.tools`
- All Messages are in `langchain.messages`
- All Embeddings are in `langchain.embeddings`

LangChain v1: What's New

Major Changes in v1.0 (October 2025):

- **Simplified Package Structure:**
 - Core functionality in `langchain`
 - Legacy features moved to `langchain-classic`
 - Requires Python 3.10+ (Python 3.9 EOL October 2025)
- **Agent-First Design:**
 - New `create_agent()` function
 - Built on LangGraph runtime
 - Durable execution and persistence
- **Content Blocks:**
 - Standardized message format across providers
 - Support for text, images, reasoning, tool calls
 - Better multimodal support
- **Middleware System:**
 - Inject custom logic at any point
 - Human-in-the-loop support
 - Error handling and retries

(Ref: LangChain v1.0 Release Notes)

How LangChain 1.0 Simplifies Agents to Just 10 Lines of Code

- LangChain 1.0 Alpha released; full 1.0 expected by late October.
- Rewritten as a simplified agent runtime built on LangGraph.
- Agents can now be created in roughly 10/few lines of code.
- Legacy LangChain moved to "LangChain Classic."
- Functionally similar to OpenAI or Pedantic SDK agents.
- Unified "create_agent" abstraction across languages.
- Integrates easily with external SDKs like Google's 80k.

Why LangGraph 1.0 is Nearly Non-Breaking and Production-Ready

- LangGraph 1.0 introduces minimal breaking changes from prior releases.
- Existing LangGraph implementations continue to work as-is.
- Core runtime stable across Python and TypeScript.
- Supports durable execution with checkpoints and rollback.
- State can persist via Postgres or SQLite for reliability.
- Human-in-the-loop and interrupt handling built-in.
- Ready for production-grade use; used by Uber, LinkedIn, Klarna, JPMorgan, Cloudflare.

The New Standardized Content Blocks for Easier Model Switching

- LangChain Core now uses standardized content/message blocks.
- Abstracts input/output across providers like OpenAI and Anthropic.
- Simplifies switching models without rewriting logic.
- Unifies multimodal inputs, reasoning traces, and tool calls.
- Middleware layer ensures consistent formatting and metadata.
- Supports normalized logging across different model providers.
- Key enabler for multi-model experimentation and portability.

Durable Execution, Streaming, Human-in-the-Loop, and Time Travel

- Agents persist state with rollback and checkpointing.
- Supports human approval and manual intervention mid-run.
- Time travel enables returning to earlier workflow states.
- Four streaming modes: messages, updates, values, custom.
- Update streaming ideal for dashboards and UX refresh.
- State persistence possible locally or via Postgres.
- Enables retry and branch execution for debugging and auditing.

Summary: What's New in LangChain Ecosystem

(Oct 2025, Release of 1.0 version)

1. LangChain Agents: The Standard Loop

- **High-Level Abstraction:** Simplest and fastest way to build an agent.
- **Primary Pattern:** Implements the standard **ReAct-style loop** (Reasoning + Acting).
- **Key Feature:** The `create_agent` function (introduced in v1.0) is the simplest and fastest way to build a production-ready agent. It abstracts away the complexity.
- **Process:** The LLM decides at each step whether to use a tool or provide a final answer.
- **Underlying Technology:** LangChain's agents since v1.0 are built **on top of the LangGraph runtime**.
- **Use Case:** Quick start, simple linear tasks, default agent behavior.

2. LangGraph: Workflow Automation & Control

- **Role:** Low-level **orchestration framework** and runtime.
- **Core Feature:** Defines explicit workflow automation and agent orchestration. Complex, stateful, and cyclic workflows as directed graphs.
- **Structure:** You define:
 1. **Nodes:** Steps (LLM call, tool execution, custom function).

- Edges:** Conditional logic for transitions (e.g., if tool succeeds, go to Node B; if it fails, go to Node C).
- Multi-Agent Systems:** Provides the explicit control needed for building supervised multi-agent systems and defining handoffs.
- Relationship to LangChain:** Used when the standard LangChain agent loop is insufficient, requiring **fine-grained control** over the flow.

3. Deep Agents: Concept for Complex Tasks

- Concept/Library:** Not a separate module, but an architectural pattern built on LangGraph for solving highly complex, long-running tasks.
- Focus:** Advanced context management and planning.
- Key Techniques:**

- Planning/Decomposition:** Breaking down the goal (e.g., using a "Todo List" tool).
 - Sub-agents:** Delegating specialized tasks to other agents for parallel or sequential execution.
 - Long-Term Memory:** Utilizing external memory or file systems for persistent context.
- Underlying Runtime:** Deep Agents are also built on LangGraph, leveraging its capabilities to manage the complex state and flow of the agent's multi-step plan.

Summary: Agent Framework Comparison

Key Takeaway: LangGraph provides the underlying runtime and control flow for both simple and complex agent applications.

Conclusions

Conclusions

Key Takeaways

What We Learned:

- LCEL makes chains simple
- Agents enable autonomy
- RAG connects to your data
- Memory maintains context
- Modular design = flexibility

You're ready to build intelligent LLM applications!

LangChain at a Glance



- Models:** LLMs, Chat, Embeddings
- Prompts:** Dynamic templates
- Chains:** Compose with LCEL
- Memory:** Conversation state
- Retrievers:** Document access
- Agents:** Autonomous tools

(Ref: Building Generative AI applications - Anand Iyer, Rajesh Thallam)

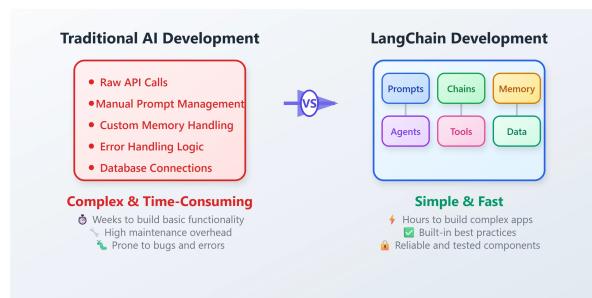
Best Practices to Remember

Development:

- Use LCEL for chains
- Start with simple models
- Add complexity gradually
- Test with small datasets

Production:

- Enable LangSmith tracing
- Implement error handling
- Monitor token usage
- Use async for scale



(Ref: What is LangChain and Why Should You Care? - Saif Ali)

Golden Rule: Keep it simple, make it work, then optimize

Resources & Next Steps

Official Documentation:

- LangChain Docs: <https://python.langchain.com>
- LangChain Blog: <https://blog.langchain.dev>
- LangChain Academy: <https://academy.langchain.com>
- API Reference: <https://api.python.langchain.com>

GitHub Repositories:

- Core: <https://github.com/langchain-ai/langchain>
- Templates: <https://github.com/langchain-ai/langchain/tree/master/templates>
- LangGraph: <https://github.com/langchain-ai/langgraph>

Community:

- Discord: <https://discord.gg/langchain>
- Twitter: @LangChainAI
- YouTube: LangChain official channel

Practice:

- Start with simple LCEL chains
- Build a RAG application
- Create custom tools and agents
- Deploy with LangServe

Final Thoughts



LangChain makes building with LLMs accessible

- Start experimenting today
- Join the community
- Share your projects
- Keep learning and building

Thank you! Questions? Let's discuss.

(Ref: What is Langchain and why should I care as a developer? - Logan Kilpatrick)

References

References

Many publicly available resources have been refereed for making this presentation. Some of the notable ones are:

- Intro to LangChain for Beginners - A code-based walkthrough- Menlo Park Lab
- LangChain Crash Course (10 minutes): Easy-to-Follow Walkthrough of the Most Important Concepts - Menlo Park Lab
- Official doc: <https://docs.langchain.com/docs/>
- Git Repo: <https://github.com/hwchase17/langchain>

- LangChain 101: The Complete Beginner's Guide Edrick <https://www.youtube.com/watch?v=P3MABz2eMUI>
A wonderful overview, don't miss
- Cookbook by Gregory Kamradt(Easy way to get started): <https://github.com/gkamradt/langchain-tutorials/blob/main/LangChain%20Cookbook.ipynb>
- Youtube Tutorials: https://www.youtube.com/watch?v=_v_fgW2SkkQ
- LangChain 101 Course (updated with LCEL) - Ivan Reznikov
- <https://github.com/IvanReznikov/DataVerse/tree/main/Courses/LangChain>
- A Complete LangChain Guide <https://nanonets.com/blog/langchain/python>
- 7 Ways to Use LangSmith's Superpowers to Turbo-boost Your LLM Apps - Menlo Park Lab
- LangChain official blog: <https://blog.langchain.dev>
- LangChain Academy: <https://academy.langchain.com>
- LangChain templates repo: <https://github.com/langchain-ai/langchain/tree/master/templates>
- Groq Documentation: <https://console.groq.com/docs>
- Groq Model Playground: <https://groq.com/>
- LangChain Groq Integration: <https://python.langchain.com/d>
- Groq GitHub Examples: <https://github.com/groq/groq->