LangChain

Abstraction layers and application development

Abstraction Layers for LLMs

Why We Need Abstraction Layers

The Challenges They Solve

- **Provider lock-in**: Dependency on a single AI vendor
- **API inconsistency**: Different interfaces across providers
- **Complexity management**: Repetitive prompt engineering
- **Context handling**: Managing token limits and conversations
- **Reusable patterns**: Reimplementing common workflows

Key Benefits

- Switch between AI providers with minimal code changes
- Simplify complex operations with pre-built components
- Standardize interfaces for consistent development
- Implement proven patterns without starting from scratch
- Build production-ready features more efficiently

The Abstraction Layer Ecosystem

LangChain

Comprehensive framework with 107K+ GitHub stars. Offers chains, agents, RAG, and extensive integrations.

LlamaIndex

Data connection specialist with 41K+ stars. Focuses on document ingestion and retrieval.

LiteLLM

Provider standardization with 19K+ stars. Unified interface for 100+ LLM APIs.

Semantic Kernel

Microsoft's enterprise solution with 24K+ stars. Strong in .NET environments.

DSPy

Research-backed framework with 24K+ stars. Focuses on programming rather than prompting.

Instructor

Structured output specialist with 10K+ stars. Built on Pydantic for validation.

Spectrum of AI Abstraction

Low-Level Abstraction

API standardization and basic utilities

LiteLLM Instructor

Mid-Level Abstraction

Chains, RAG patterns, and data integration

LangChain LlamaIndex Haystack

High-Level Abstraction

Agent orchestration and autonomous systems

LangGraph AutoGen Semantic Kernel

Introduction to LangChain

Why We're Focusing on LangChain

Strategic Advantages

- **Most popular** with 107K+ GitHub stars
- Most **extensive integrations** ecosystem
- Support for both **Python and JavaScript/TypeScript**
- Complete end-to-end application capabilities
- **Strong community** and frequent updates
- Used by Fortune 100 companies at scale

Comprehensive Solution - Covers from basic prompting to complex agents

Production Readiness - Built-in features for monitoring, observability, and deployment

Transferable Skills - Concepts you learn apply to many other frameworks

Industry Standard - Most widely used in job descriptions and enterprise settings

LangChain Concepts Reference

- **Chat models** *
- **LCEL** *
- **Chat history** *
- Multimodality
- Output parsers
- **Prompt templates** *
- Retrieval
- Agents
- Document loaders
- Few-shot prompting
- Example selectors
- RAG
- Runnable interface
- Tools
- Architecture
- Embedding models
- Text splitters
- **Messages** *
- **Streaming** *
- Structured output
- **Why LangChain?** *

Chat Models

What Are Chat Models?

- Advanced LLMs that take **message lists** as input
- Return structured message responses
- Standard interface across model providers
- Support for async and streaming operations

Key Features

- Native tool calling capabilities
- Structured output generation
- Consistent parameter controls
- Multimodal support (images, audio, video)

```python
from langchain_openai import ChatOpenAI
from typing import List
from pydantic import BaseModel

model = ChatOpenAI(
    model="gpt-4o",
    temperature=0.7
)

response = model.invoke([
    {"role": "system", "content": "You are helpful."},
    {"role": "user", "content": "Hello world!"}
])

class Movie(BaseModel):
    title: str
    year: int

movies = model.with_structured_output(List[Movie]).invoke(
    "List 3 sci-fi movies from the 1980s"
)
```

LangChain Expression Language (LCEL)

What Is LCEL?

- Declarative way to build chains
- Uses the `|` operator to compose components
- Makes data flow visible and intuitive
- "Describe what, not how"

Key Benefits

- Optimized parallel execution
- Built-in streaming support
- Seamless async support
- Simplified error handling

```python
# Import components
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

# Create prompt
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a poem generator."),
    ("user", "Write a poem about {topic}")
])

# Initialize model
model = ChatOpenAI()

# Create chain using LCEL pipe operator
chain = prompt | model | StrOutputParser()

# Execute the chain
result = chain.invoke({"topic": "artificial intelligence"})
```

Chat History

What Is Chat History?

- Record of conversation messages
- Maintains context between interactions
- Structured system for message management
- Enables coherent, contextual exchanges

Key Considerations

- Must respect model context window limits
- Requires structured message formatting
- Should follow proper role sequencing
- Enables personalized, continuous interactions

```python
from langchain_core.messages import HumanMessage, AIMessage
from langchain_core.prompts import ChatPromptTemplate, MessagesPlace
from langchain_openai import ChatOpenAI
model = ChatOpenAI()

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

chat_history = [
    HumanMessage(content="Hello, who are you?"),
    AIMessage(content="I'm an AI assistant here to help.")
]

chain = prompt | model

response = chain.invoke({
    "chat_history": chat_history,
    "input": "What can you help me with?"
})
```

Prompt Templates

What Are Prompt Templates?

- Structured patterns for LLM inputs
- Standardize communication with models
- Support variable interpolation
- Enable consistent, reusable prompting

Types of Templates

- **String Templates**: Simple variable replacement
- **Chat Templates**: Structure multi-role conversations
- **Placeholder**: Dynamically insert message lists
- **FewShot Templates**: Include examples for learning

```python
from langchain_core.prompts import PromptTemplate, ChatPromptTemplat

# String template
basic_prompt = PromptTemplate.from_template(
    "Write a {adjective} poem about {subject}."
)

# Chat template
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a {role} assistant."),
    ("user", "Help me with {task}."),
    ("assistant", "I'll help you with {task}."),
    ("user", "{follow_up}")
])

# Using a template
result = basic_prompt.format(
    adjective="whimsical",
    subject="machine learning"
)
```

Messages

What Are Messages?

- Fundamental unit of chat communication
- Structured by role and content
- Standard format across different models

Message Types

- **SystemMessage**: Set context and behavior
- **HumanMessage**: User inputs
- **AIMessage**: Model responses
- **ToolMessage**: Tool call results
- **AIMessageChunk**: Streaming response pieces

```
from langchain_core.messages import (
    SystemMessage,
    HumanMessage,
    AIMessage,
    ToolMessage
)
from langchain_openai import ChatOpenAI

messages = [
    SystemMessage(content="You are a helpful assistant."),
    HumanMessage(content="What's the weather today?"),
    AIMessage(content="I don't have access to real-time data."),
    HumanMessage(content="Can you check?"),
    ToolMessage(
        content="72°F, Sunny",
        tool_call_id="weather_tool",
        name="weather_tool"
    )
]
model = ChatOpenAI()
response = model.invoke(messages)
```

Streaming

What Is Streaming?

- Progressive output delivery from LLMs
- Reduces perceived latency
- Provides real-time feedback

Key Features

- Works with all major model providers
- Supports both sync and async patterns
- Can stream individual tokens or chunks
- Automatically enabled in many workflows
- Works with complex chains and graphs

```python
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

# Create components
model = ChatOpenAI(streaming=True)
prompt = ChatPromptTemplate.from_messages([
    ("system", "You write detailed explanations."),
    ("user", "Explain {topic} in detail")
])

# Create chain
chain = prompt | model

# Synchronous streaming
for chunk in chain.stream({"topic": "quantum computing"}):
    print(chunk.content, end="", flush=True)

# Asynchronous streaming
async for chunk in chain.astream({"topic": "AI ethics"}):
    # In a web app, send each chunk to the client
    print(chunk.content, end="", flush=True)
```

Python vs. JavaScript Implementation

Python

```python
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

# Create a model
model = ChatOpenAI(model="gpt-4o-mini")

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

# Create and use a chain
chain = prompt | model
response = chain.invoke({"input": "Hello!"})
print(response.content)
```

JavaScript/TypeScript

```javascript
import { ChatOpenAI } from "@langchain/openai";
import { ChatPromptTemplate } from "@langchain/core/prompts";

// Create a model
const model = new ChatOpenAI({ model: "gpt-4o-mini" });

// Create a prompt template
const prompt = ChatPromptTemplate.fromMessages([
    ["system", "You are a helpful assistant."],
    ["user", "{input}"]
]);

// Create and use a chain
const chain = prompt.pipe(model);
const response = await chain.invoke({ input: "Hello!" });
console.log(response);
```

Building Chat Interfaces with Streamlit

What is Streamlit?

CW
JV

Streamlit Essentials

- Python-first web app framework
- Built for data scientists and ML engineers
- Turn Python scripts into interactive web apps
- No frontend experience required

Key Features

- Extremely fast prototyping
- Rich interactive components
- Automatic UI updates on code changes
- Built-in chat interface components

Rapid Development - Create AI interfaces in minutes not days

Chat Components - Purpose-built for conversational AI

LLM Integration - Works seamlessly with LangChain, Gemini, and others

Free Deployment - Share apps with others via Streamlit Community Cloud

Streamlit Chat Interface Code Example

Initialize Chat State

```python
import streamlit as st
from langchain_core.messages import AIMessage, HumanMessage

st.title("AI Chatbot")
st.caption("Powered by LangChain and Streamlit")

if "chat_history" not in st.session_state:
    st.session_state.chat_history = [
        AIMessage(content="How can I help you today?")
    ]

for message in st.session_state.chat_history:
    if isinstance(message, AIMessage):
        with st.chat_message("assistant"):
            st.write(message.content)
    elif isinstance(message, HumanMessage):
        with st.chat_message("user"):
            st.write(message.content)
```

Handle User Input

```python
if user_input := st.chat_input("Type your message..."):
    st.session_state.chat_history.append(
        HumanMessage(content=user_input)
    )

    with st.chat_message("user"):
        st.write(user_input)

    with st.chat_message("assistant"):
        # This is where you'd normally call your LLM
        response = "Nice message! This is a placeholder"

        # Add AI response to history
        st.session_state.chat_history.append(
            AIMessage(content=response)
        )

        # Display the response
        st.write(response)
```

Integrating LangChain with Streamlit

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import MessagesPlaceholder

#... create prompt and llm
chain = prompt | llm

if user_input := st.chat_input("Type something..."):
    chat_history = []
    for msg in st.session_state.messages:
        if msg["role"] == "user":
            chat_history.append(HumanMessage(content=msg["content"])
        else:
            chat_history.append(AIMessage(content=msg["content"]))

    with st.chat_message("assistant"):
        response = chain.invoke({
            "chat_history": chat_history,
            "input": user_input
        })
        st.write(response.content)
```

Key Integration Points

- LangChain provides the LLM interaction layer
- Streamlit provides the UI components
- Session state maintains conversation history
- Messages are converted between formats as needed

Best Practices

- Initialize the app state early
- Use streaming for better UX with longer responses
- Structure your app for maintainability
- Add error handling for API failures

Advanced Streamlit Features

UI Enhancements

- **st.sidebar**: Add configuration controls
- **st.expander**: Collapsible sections
- **st.tabs**: Organize content into tabs
- **st.columns**: Multi-column layouts
- **st.file_uploader**: Enable document analysis

State Management

- **st.session_state**: Persistent app state
- **st.cache_data**: Cache expensive operations
- **st.cache_resource**: Cache model loading
- **st.form**: Bundle related inputs
- **st.status**: Show operation progress

Streaming Example

```python
from langchain_openai import ChatOpenAI
import streamlit as st

model = ChatOpenAI(streaming=True)

st.title("Streaming Demo")

if user_input := st.chat_input("Ask something..."):
    with st.chat_message("user"):
        st.write(user_input)

    with st.chat_message("assistant"):
        message_placeholder = st.empty()
        full_response = ""

        # Stream the response
        for chunk in model.stream(user_input):
            full_response += chunk.content
            message_placeholder.markdown(full_response + "▌")

        message_placeholder.markdown(full_response)
```

When to Use Streamlit

- Rapid AI prototyping and demos
- Data-driven applications
- Internal tools and dashboards
- Research and experimentation

Have Fun!

Continue learning at LangChain Documentation and Streamlit Documentation