
LangChain.js

¿Qué es LangChain?

Framework para desarrollar aplicaciones con Modelos de Lenguaje (LLMs)

Conectando LLMs con el mundo real

Historia de LangChain

Octubre 2022: Nace LangChain en **Python** 🐍

Creado por Harrison Chase • Enfoque inicial: encadenar prompts y LLMs

Diciembre 2022: Lanzamiento de **LangChain.js** ⚡

Apenas 2 meses después • Port oficial • Para desarrolladores web

2023: Explosión de adopción en ambos ecosistemas

Múltiples proveedores de LLMs • 110K+ stars en GitHub

2024-2025: Maduración hacia producción

LangGraph: Control de bajo nivel • **LangSmith**: Observabilidad y testing

Historia de LangChain

- **2022**: Nace LangChain - Abstracciones básicas de "chains"
  - Enfoque inicial: encadenar prompts y llamadas a LLMs
- **2023**: Explosión de adopción y evolución
  - Integración con múltiples proveedores de LLMs
  - Comunidad activa y rápido crecimiento
- **2024-2025**: Maduración hacia producción
  - **LangGraph**: Control de bajo nivel para agentes complejos
  - **LangSmith**: Observabilidad y debugging
  - Enfoque en fiabilidad para entornos de producción

LangChain: Python y JavaScript

Dos implementaciones, misma filosofía

🐍 LangChain Python

El hermano mayor (2022)

- Primera implementación
- Más maduro y completo
- Ideal para data science y ML
- 110K+ GitHub stars

⚡ LangChain.js

Para desarrolladores web

- TypeScript/JavaScript nativo
- Paridad de funcionalidades
- Frontend y backend
- 130M+ descargas totales

🔄 Mismo concepto, diferentes ecosistemas

Elige según tu stack tecnológico

Filosofía de LangChain

Flexibilidad

Sin vendor lock-in

Integración

LLMs + Datos externos

Agentes

Decisiones autónomas

Producción

Del prototipo a la realidad

Principios Fundamentales

1. Flexibilidad de Modelos

Estandarización de inputs/outputs entre proveedores de LLMs. Cambia fácilmente entre OpenAI, Anthropic, Google, etc.

2. Orquestación Compleja

Los modelos coordinan flujos de trabajo con datos externos y herramientas. Más allá de la simple generación de texto.

3. Fiabilidad en Producción

Observabilidad, testing y debugging para aplicaciones reales. El prototipo es fácil, producción es el reto.

Creencias Fundamentales

🚀 Los LLMs son tecnología poderosa

Una innovación que vale la pena construir sobre ella

🔗 Los LLMs son mejores con datos externos

Combinar LLMs con fuentes de datos externas potencia sus capacidades

🤖 El futuro es agéntico

Las aplicaciones exhibirán capacidades autónomas y de toma de decisiones

🌱 Estamos en una etapa temprana

El campo está en constante evolución con espacio sustancial para crecer

💪 Producción es difícil

Prototipar es fácil, lograr fiabilidad en producción es el verdadero desafío

Ecosistema LangChain

LangChain

Framework core
Chains, Prompts, LLMs

LangGraph

Agentes complejos
Control de bajo nivel

LangSmith

Observabilidad
Debug y monitorización

¿Quién usa LangChain?

🏢 Empresas en Producción

- **LinkedIn** - Agentes con LangGraph
- **Uber** - Sistemas agénticos
- **Replit** - Asistentes de código
- **Klarna** - Chatbots de atención
- **Snowflake** - Integración de datos
- **J.P. Morgan** - Aplicaciones financieras
- **BCG** - Consultoría con IA

📊 Números de Adopción

- **1,306** empresas verificadas
- **132K+** aplicaciones LLM construidas
- **130M+** descargas totales
- **51%** usando agentes en producción
- **78%** planean usar agentes pronto

💼 Casos de Uso Comunes

- Chat sobre documentos
- Atención al cliente
- Asistentes personalizados
- Análisis de datos

Modelo de Negocio de LangChain

🏢 La Empresa

- **Fundadores**: Harrison Chase y Ankush Gola
- **Fundación**: Octubre 2022
- **Sede**: San Francisco, CA
- **CEO**: Harrison Chase
- **Equipo**: ~136 empleados (2025)

💰 Financiación

- **$260M** levantados en total
- **$1.25B** de valoración (Oct 2025)
- **Inversores**: Sequoia, Benchmark, IVP, CapitalG

💼 Modelo de Negocio

Freemium B2B

- Framework open-source (gratis)
- Monetización vía LangSmith

Revenue 2025: $16M ARR

- Crecimiento: $8.5M → $16M en 1 año
- **1,000+** clientes empresariales

💎 LangSmith Pricing

- **Developer**: Gratis (5k traces/mes)
- **Plus**: $39/usuario/mes (10k traces)
- **Enterprise**: Personalizado + self-hosted

LangChain.js

La implementación JavaScript/TypeScript del framework LangChain

¿Por qué LangChain.js?

- **Desarrollo web nativo**
  - Frontend y backend con la misma tecnología
- **Ecosistema JavaScript**
  - NPM, TypeScript, Node.js
- **Paridad con Python**
  - Mismos conceptos y APIs
- **Comunidad activa**
  - Mantenido oficialmente

LangChain.js: Entornos de Ejecución

🖥️ Runtimes Soportados

- **Node.js** (20.x, 22.x, 24.x)
  - ESM y CommonJS
- **Deno** - Runtime seguro
- **Bun** - Runtime ultra-rápido
- **Navegadores** - Client-side

☁️ Edge & Serverless

- **Cloudflare Workers**
- **Vercel Edge Functions**
- **Next.js** (Server, Edge, Browser)
- **Supabase Edge Functions**

💪 Ventaja Clave

Una sola codebase para frontend, backend, edge y serverless

Arquitectura de Componentes

Los componentes trabajan juntos para crear aplicaciones AI sofisticadas

🧩 Seis Categorías Principales

📝 Models

Chat, LLMs, embeddings

🔧 Tools

APIs, búsquedas, DBs

🤖 Agents

Orquestación y razonamiento

💾 Memory

Historia y estado

🔍 Retrievers

Búsqueda vectorial y web

📄 Documents

Loaders, splitters

Pipeline de Procesamiento

Flujo de datos en 5 capas

1. 📥 **Input Processing** Datos raw → documentos
2. 🔢 **Embedding & Storage** Texto → vectores semánticos
3. 🔍 **Retrieval** Búsqueda de información relevante
4. ✨ **Generation** LLM genera respuestas + tools
5. 🎯 **Orchestration** Agentes coordinan el flujo

🏗️ Patrones Comunes

RAG

Retrieval Augmented Generation

Agentes

Decisiones autónomas con tools

Multi-agente

Sistemas coordinados especializados

Estructura del Curso

LangChain.js: De cero a producción

Tu hoja de ruta completa

Módulos del Curso

📚 Módulo 1: Introducción

Qué es LangChain, filosofía, historia y ecosistema

⚙️ Módulo 2: Instalación y Setup

Configuración, APIs, proveedores de LLMs

🤖 Módulo 3: Modelos

Chat Models, LLMs, parámetros, streaming

Módulos del Curso

💬 Módulo 4: Mensajes y Prompts

Sistema de mensajes, templates, contexto, historial

🎯 Módulo 5: Agentes

Tools, structured output, middlewares, runtime, human-in-the-loop

🔍 Módulo 6: LangSmith

Observabilidad, debugging, evaluación y producción

Progresión del Aprendizaje

Fundamentos

Módulos 1-2

Instalación y conceptos básicos

Construcción

Módulos 3-4

Modelos, mensajes y prompts

Avanzado

Módulos 5-6

Agentes y producción

Al Final del Curso

🎓 Serás capaz de:

✅ Crear aplicaciones LLM completas

✅ Implementar agentes autónomos

✅ Gestionar conversaciones con contexto

✅ Desplegar en producción con confianza

Instalación y Setup

Módulo 2: Primeros Pasos con LangChain.js

Configurando tu entorno de desarrollo

LangChain v1.x: Arquitectura de Paquetes

🎯 Cambio importante: Estructura Modular

❌ **v0.x (Deprecated)**

```
langchain/chat_models
```

✅ **v1.x (Actual)**

```
@langchain/openai
```

📦 Paquetes Principales

@langchain/core

Interfaces base

@langchain/langgraph

Agentes

@langchain/openai

OpenAI/Azure

@langchain/anthropic

Claude

@langchain/google-genai

Gemini

langchain

Meta-package

Instalación en Node.js

Requisitos: Node.js v20+, npm/pnpm/yarn/bun

🚀 Instalación Básica

```
# Paquete core (obligatorio)
npm install @langchain/core

# Proveedor de LLM (ej: OpenAI)
npm install @langchain/openai
```

💡 Instalación Completa

```
# Todo lo esencial
npm install @langchain/core @langchain/openai @langchain/langgraph

# TypeScript (si no lo tienes)
npm install -D typescript @types/node
```

📌 Versiones Actuales (Enero 2025)

- `@langchain/core` : **v1.1.11**
- `@langchain/langgraph` : **v1.0.7**

Configuración del Entorno

🔑 Variables de Entorno (.env)

```
# Crea un archivo .env en la raíz
OPENAI_API_KEY=sk-proj-...
ANTHROPIC_API_KEY=sk-ant-...
```

📦 Opción A: dotenv (tradicional)

```
npm install dotenv
```

```
import 'dotenv/config';  // Al inicio del archivo
```

⚡ Opción B: --env-file (Node.js 20+)

```
# Sin instalar nada!
npx tsx --env-file=.env index.ts
```

✅ Recomendado: Sin dependencias adicionales

API Key en el Constructor

🔧 Pasar la key directamente

```
import { ChatOpenAI } from "@langchain/openai";

const model = new ChatOpenAI({
  apiKey: "sk-proj-xxxxxxxxxxxx",  // Directamente aquí
  model: "gpt-4o-mini"
});
```

⚠️ Cuándo usarlo

✅ Variables de entorno

Para producción, CI/CD y equipos

⚡ Constructor

Solo testing rápido y prototipos

🔒 Recuerda: Añade `.env` a `.gitignore`

Hello World con LangChain.js

Tu primera aplicación LLM

```ts
// index.ts
import { ChatOpenAI } from "@langchain/openai";

// 1. Crear el modelo
const model = new ChatOpenAI({
  model: "gpt-4o-mini",
});

// 2. Invocar el modelo
const response = await model.invoke(
  "¿Qué es LangChain?"
);

// 3. Ver la respuesta
console.log(response.content);
```

🏃 Ejecutar con tsx

```bash
# Instalar tsx
npm install -D tsx

# Ejecutar directamente
npx tsx index.ts
```

📤 Salida Esperada

```
LangChain es un framework para
desarrollar aplicaciones con LLMs.
Permite conectar modelos de lenguaje
con datos externos y herramientas...
```

🎯 Conceptos Clave

- **Model**: ChatOpenAI
- **invoke()**: Llamada síncrona
- **response.content**: El texto generado

Ejemplo con Google Gemini

Alternativa gratuita a OpenAI

```ts
// gemini-example.ts
import { ChatGoogleGenerativeAI } from "@langchain/google-g

// 1. Crear el modelo
const model = new ChatGoogleGenerativeAI({
  model: "gemini-2.0-flash-exp",
});

// 2. Invocar
const response = await model.invoke(
  "¿Qué es LangChain?"
);

console.log(response.content);
```

📦 Instalación

```bash
npm install @langchain/google-genai
```

🔑 API Key

```
# .env
GOOGLE_API_KEY=AI...
```

Obtener gratis en: ai.google.dev/gemini-api

⚡ Ejecutar

```bash
npx tsx --env-file=.env gemini-example.ts
```

¿Qué LLM usar?

Comparativa de proveedores principales

Proveedor	Modelo	Empresa	Free Tier	Precio API*
OpenAI	GPT-4o	USA	❌ No	$5/$15
Google	Gemini 2.0 Flash	USA	✅ Sí	$1.25/$10
Anthropic	Claude 4 Sonnet	USA	✅ Límite diario	$3/$15
Mistral	Mistral Large	Francia (UE)	✅ Experiment	~$2/$6
DeepSeek	DeepSeek R1	China	✅ Demo gratis	$0.55/$2.19

* Precio por millón de tokens (input/output) - Enero 2025

⚠️ Considera las implicaciones de privacidad según tu jurisdicción y tipo de datos

URLs para obtener API Keys

OpenAI

platform.openai.com

Google Gemini

ai.google.dev/gemini-api

Anthropic (Claude)

console.anthropic.com

Mistral AI

console.mistral.ai

DeepSeek

platform.deepseek.com

Ollama (Local)

ollama.com

Recomendaciones: Para Empezar

🎓 Para aprender y empezar

Google Gemini o Mistral Experiment

✅ Gratis

✅ Sin tarjeta de crédito

✅ Límites generosos para desarrollo

Recomendaciones: Producción

💼 Para producción

Claude 4 Sonnet o GPT-4o

✅ Mejor precisión y razonamiento

⚠️ Requiere presupuesto ($3-15/M tokens)

💰 Alto volumen / Económico

DeepSeek R1

✅ 90% más barato ($0.55-$2.19/M)

Estrategia Recomendada

🏠 Desarrollo: Ollama (Local)

Para testing, iteración rápida y privacidad

✅ Gratis • Sin límites • Privado

⚠️ Requiere 16GB RAM mínimo

☁️ Producción: APIs Cloud

Para usuarios finales

✅ Mayor calidad • Escalable • Fiable

Ollama: Requisitos de Hardware

📊 ¿Qué necesitas según el modelo?

Tamaño	RAM	VRAM GPU	Modelos Populares
3B	8GB	4GB	Llama 3.2 3B, Phi-3 Mini
7B	16GB	8GB	Llama 3.1 7B, Mistral 7B
13B	32GB	16GB	Llama 3.1 13B
70B	64GB+	24GB+	Llama 3.1 70B

💡 Recomendaciones

🎓 Para empezar

Llama 3.2 3B

8GB RAM suficiente

💻 Desarrollo

Llama 3.1 7B

16GB RAM + GPU

🏢 Producción

Usa APIs cloud

Más fiable

Usar Ollama con LangChain.js

📦 Instalación

```
# 1. Descargar e instalar Ollama
# https://ollama.com/download

# 2. Descargar un modelo
ollama pull llama3.2

# 3. Instalar paquete LangChain
npm install @langchain/ollama
```

💻 Código

```
import { ChatOllama } from "@langchain/ollama";

const model = new ChatOllama({
  model: "llama3.2",
  baseUrl: "http://localhost:11434",
});

const response = await model.invoke("¿Qué es LangChain?");
console.log(response.content);
```

¿Qué es Zod?

Zod es una librería de validación de esquemas para TypeScript

¿Para qué lo usamos en LangChain?

- ✅ Definir la estructura esperada de las respuestas del LLM
- ✅ Validar que el LLM devuelve datos en el formato correcto
- ✅ Obtener autocompletado y type safety en TypeScript
- ✅ Convertir respuestas de texto a objetos JavaScript tipados

```
npm install zod
```

Ejemplo de Zod

```ts
import { z } from "zod";

// Definir el esquema de datos
const UserSchema = z.object({
  name: z.string(),
  age: z.number().min(0).max(120),
  email: z.string().email(),
  roles: z.array(z.enum(["admin", "user", "guest"])),
});

// Inferir el tipo TypeScript automáticamente
type User = z.infer<typeof UserSchema>;

// Validar datos
const result = UserSchema.parse({
  name: "Juan",
  age: 30,
  email: "juan@example.com",
  roles: ["admin", "user"]
});
// ✅ Si los datos son válidos, devuelve el objeto tipado
// ❌ Si no, lanza un error con detalles del problema
```

Módulo 3: Modelos

Trabajando con LLMs y Chat Models en LangChain.js

Chat Models vs LLMs

💬 Chat Models

Diseñados para conversaciones

Entrada: Mensajes (System, Human, AI)

Salida: Mensajes estructurados

Ejemplos: GPT-4o, Claude, Gemini

🤖 LLMs (Legacy)

Modelos tradicionales

Entrada: Texto plano (string)

Salida: Texto plano (string)

Ejemplos: GPT-3.5 Instruct

✅ **Recomendación:** Usa siempre Chat Models (son el estándar actual)

¿Qué es un Token?

Token = Unidad básica de texto que procesa un LLM

No es exactamente una palabra, ni un carácter

Ejemplos de Tokenización

`"Hola mundo"` → **2 tokens**

`"LangChain"` → **2 tokens** (Lang + Chain)

`"desarrollador"` → **3 tokens**

`"🚀"` → **1-2 tokens**

Regla General

1 token ≈ **4 caracteres** en inglés

1 token ≈ **¾ de palabra** en inglés

100 tokens ≈ **75 palabras**

¿Cómo se Cuentan los Tokens?

Los tokens se cuentan en **entrada (input)** y **salida (output)** por separado

💰 Costos

Los LLMs cobran por tokens procesados

Output suele ser **más caro** que input (3-5x)

Ejemplo: GPT-4o → $2.50/1M tokens input, $10/1M tokens output

🔧 Herramienta para Contar Tokens

OpenAI Tokenizer (funciona para la mayoría de modelos):

Parámetros Principales

🌡️ temperature

Controla la aleatoriedad de las respuestas

0.0 = Determinista | 1.0 = Creativo | 2.0 = Muy aleatorio

📏 maxTokens

Límite máximo de tokens en la respuesta

Ejemplo: 500 (aprox. 375 palabras)

Parámetros Principales

🎯 topP

Nucleus sampling (alternativa a temperature)

0.1 = Conservador | 0.9 = Diverso

🔁 n

Número de respuestas a generar

Útil para comparar variaciones

Ejemplo: Configurando Parámetros

```
import { ChatOpenAI } from "@langchain/openai";

const model = new ChatOpenAI({
  model: "gpt-4o-mini",
  temperature: 0.7,        // Balance entre creatividad y coherencia
  maxTokens: 500,          // Limitar longitud de respuesta
  topP: 0.9,              // Nucleus sampling
});

const response = await model.invoke(
  "Explica qué es un agente en LangChain"
);

console.log(response.content);
```

💡 **Tip:** Usa `temperature: 0` para respuestas deterministas (útil en tests)

Streaming: Respuestas en Tiempo Real

¿Por qué streaming?

- ✅ Experiencia de usuario mejorada (como ChatGPT)
- ✅ Feedback inmediato (no esperar 10-20 segundos)
- ✅ Procesar mientras se genera

```
import { ChatOpenAI } from "@langchain/openai";

const model = new ChatOpenAI({
  model: "gpt-4o-mini",
  streaming: true,
});

const stream = await model.stream("Explica qué es LangChain");

for await (const chunk of stream) {
  console.log(chunk.content);  // Imprime palabra por palabra
}
```

Batch: Procesamiento en Lote

¿Cuándo usar batch?

- ✅ Procesar múltiples inputs a la vez
- ✅ Optimización de costos (algunos proveedores ofrecen descuentos)
- ✅ Mejor throughput que llamadas individuales

```
import { ChatOpenAI } from "@langchain/openai";

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

const inputs = [
  "¿Qué es un agente?",
  "¿Qué es RAG?",
  "¿Qué es LangSmith?"
];

const responses = await model.batch(inputs);

responses.forEach((response, i) => {
  console.log(`Pregunta ${i + 1}:`, response.content);
});
```

Structured Output

Problema: Las respuestas de LLMs son texto plano, difíciles de procesar

Solución: Forzar al modelo a responder con JSON estructurado

```
import { ChatOpenAI } from "@langchain/openai";
import { z } from "zod";

// Definir el esquema con Zod
const schema = z.object({
  sentiment: z.enum(["positive", "negative", "neutral"]),
  score: z.number().min(0).max(1),
  keywords: z.array(z.string()),
});

const model = new ChatOpenAI({ model: "gpt-4o-mini" })
  .withStructuredOutput(schema);

const result = await model.invoke("Me encanta LangChain!");
// { sentiment: "positive", score: 0.95, keywords: ["encanta", "langchain"] }
```

Intercambiabilidad de Modelos

Ventaja clave de LangChain: Cambiar de proveedor sin reescribir código

```
// OpenAI
import { ChatOpenAI } from "@langchain/openai";

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

```
// Anthropic Claude
import { ChatAnthropic } from "@langchain/anthropic";

const model = new ChatAnthropic({
  model: "claude-4-sonnet-20251101"
});
```

Intercambiabilidad de Modelos

```
// Google Gemini
import { ChatGoogleGenerativeAI }
  from "@langchain/google-genai";

const model = new ChatGoogleGenerativeAI({
  model: "gemini-2.0-flash-exp"
});
```

```
// Ollama (Local)
import { ChatOllama } from "@langchain/ollama";

const model = new ChatOllama({
  model: "llama3.1:7b"
});
```

⚡ Todos usan la misma interfaz: `.invoke()`, `.stream()`, `.batch()`

initChatModel: Configuración Universal

Forma simplificada de inicializar modelos sin imports específicos

```
import { initChatModel } from "langchain/chat_models/universal";

// En lugar de importar ChatOpenAI, ChatAnthropic, etc.
const model = await initChatModel("gpt-4o-mini", {
  modelProvider: "openai",
  temperature: 0.7,
});

const response = await model.invoke("Hola!");
```

✅ Ventaja: Cambiar de proveedor modificando solo el string

Providers Disponibles

OpenAI

```
modelProvider: "openai"
// gpt-4o, gpt-4o-mini, gpt-4-turbo
```

Anthropic

```
modelProvider: "anthropic"
// claude-4-sonnet, claude-4-opus
```

Google

```
modelProvider: "google-genai"
// gemini-2.0-flash-exp, gemini-pro
```

Azure OpenAI

```
modelProvider: "azure_openai"
```

Mistral

```
modelProvider: "mistral"
// mistral-large, mistral-small
```

Groq

```
modelProvider: "groq"
// llama-3.1-70b, mixtral-8x7b
```

Ollama

```
modelProvider: "ollama"
// llama3.1:7b, mistral:7b
```

Cohere

```
modelProvider: "cohere"
// command-r, command-r-plus
```

Más Providers

Fireworks

```
modelProvider: "fireworks"
```

Vertex AI (Google Cloud)

```
modelProvider: "google-vertexai"
```

Together AI

```
modelProvider: "together_ai"
```

Cloudflare

```
modelProvider: "cloudflare"
```

Bedrock (AWS)

```
modelProvider: "bedrock"
```

📚 Lista completa de providers

js.langchain.com/docs/integrations/chat

Binding Tools al Modelo

Puedes vincular tools directamente al modelo con `.bindTools()`

```
import { ChatOpenAI } from "@langchain/openai";
import { tool } from "@langchain/core/tools";
import { z } from "zod";

const weatherTool = tool(
  async ({ location }) => `El tiempo en ${location} es soleado`,
  {
    name: "get_weather",
    description: "Obtiene el tiempo de una ubicación",
    schema: z.object({ location: z.string() }),
  }
);

const model = new ChatOpenAI({ model: "gpt-4o-mini" });
const modelWithTools = model.bindTools([weatherTool]);

const response = await modelWithTools.invoke("¿Qué tiempo hace en Madrid?");
console.log(response.tool_calls);  // Tool que el LLM decidió usar
```

Diferencia: bindTools vs createAgent

🔧 .bindTools()

- El LLM decide si usar la tool
- Devuelve `tool_calls`
- Tú ejecutas la tool manualmente
- Control total del flujo

Ideal para flujos personalizados

🤖 createAgent()

- El agente decide y ejecuta
- Loop automático
- Ejecuta tools automáticamente
- Estado persistente incluido

Ideal para agentes autónomos

💡 Usa `bindTools()` para control, `createAgent()` para simplicidad

Módulo 4: Mensajes y Prompts

Sistema de mensajes, templates y gestión de contexto

Sistema de Mensajes

Los **Chat Models** trabajan con mensajes estructurados, no con texto plano

💼 SystemMessage

Define el rol y comportamiento del asistente

👤 HumanMessage

Mensaje del usuario

🤖 AIMessage

Respuesta del modelo

Ejemplo: Usando Mensajes

```
import { ChatOpenAI } from "@langchain/openai";
import {
  SystemMessage,
  HumanMessage
} from "@langchain/core/messages";

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

const messages = [
  new SystemMessage("Eres un experto en JavaScript y TypeScript"),
  new HumanMessage("¿Qué es async/await?")
];

const response = await model.invoke(messages);

console.log(response.content);
// Respuesta contextualizada como experto en JS/TS
```

Prompt Templates

Templates permiten reutilizar prompts con variables dinámicas

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

const template = ChatPromptTemplate.fromMessages([
  ["system", "Eres un experto en {topic}"],
  ["human", "{question}"]
]);

const prompt = await template.invoke({
  topic: "JavaScript",
  question: "¿Qué es async/await?"
});

const response = await model.invoke(prompt);
```

Ventajas de Templates

✅ Reutilización

El mismo template con diferentes variables

✅ Mantenibilidad

Cambiar el prompt en un solo lugar

✅ Testing

Probar con múltiples combinaciones de variables

Few-Shot Prompting

Few-shot: Dar ejemplos al LLM para que aprenda el patrón deseado

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

const template = ChatPromptTemplate.fromMessages([
  ["system", "Clasifica el sentimiento de tweets"],
  ["human", "Me encanta este producto!"],
  ["ai", "positivo"],
  ["human", "No funciona bien :("],
  ["ai", "negativo"],
  ["human", "{tweet}"]  // El tweet a clasificar
]);

const response = await model.invoke(
  await template.invoke({ tweet: "Buena experiencia" })
);
// → "positivo"
```

Historial de Conversación

Para mantener **contexto** entre mensajes, acumula el historial

```
import { ChatOpenAI } from "@langchain/openai";
import { HumanMessage, AIMessage } from "@langchain/core/messages";

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

const history = [
  new HumanMessage("Hola, me llamo Juan"),
  new AIMessage("Hola Juan, ¿en qué puedo ayudarte?"),
  new HumanMessage("¿Cuál es mi nombre?")
];

const response = await model.invoke(history);
console.log(response.content);
// → "Tu nombre es Juan"
```

Gestión de Historial

Problema: El historial crece indefinidamente → Tokens infinitos → Costos altos

📏 Limitar por cantidad

Mantener solo los últimos N mensajes

📊 Limitar por tokens

Mantener solo los últimos N tokens

📝 Resumir

Comprimir conversaciones antiguas en un resumen

Ejemplo: Limitando Historial

```
import { ChatOpenAI } from "@langchain/openai";
import { HumanMessage, AIMessage } from "@langchain/core/messages";

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

// Simular un historial largo
const fullHistory = [
  new HumanMessage("Mensaje 1"),
  new AIMessage("Respuesta 1"),
  new HumanMessage("Mensaje 2"),
  new AIMessage("Respuesta 2"),
  new HumanMessage("Mensaje 3"),
  new AIMessage("Respuesta 3"),
];

// Mantener solo los últimos 4 mensajes (2 turnos)
const recentHistory = fullHistory.slice(-4);

const response = await model.invoke([
  ...recentHistory,
  new HumanMessage("Nueva pregunta")
]);
```

Placeholder Variables

MessagesPlaceholder: Insertar arrays de mensajes dinámicamente en templates

```
import { ChatPromptTemplate, MessagesPlaceholder } from "@langchain/core/prompts";

const template = ChatPromptTemplate.fromMessages([
  ["system", "Eres un asistente útil"],
  new MessagesPlaceholder("history"),  // Array de mensajes
  ["human", "{input}"]
]);

const response = await model.invoke(
  await template.invoke({
    history: [
      new HumanMessage("Hola"),
      new AIMessage("¡Hola! ¿En qué puedo ayudarte?")
    ],
    input: "¿Qué tiempo hace?"
  })
);
```

Partial Templates

Partial: Pre-rellenar algunas variables, completar otras después

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

const template = ChatPromptTemplate.fromMessages([
  ["system", "Eres un experto en {topic} que habla {language}"],
  ["human", "{question}"]
]);

// Pre-configurar topic y language
const spanishJSTemplate = await template.partial({
  topic: "JavaScript",
  language: "español"
});

// Ahora solo necesitas la pregunta
const response = await model.invoke(
  await spanishJSTemplate.invoke({ question: "¿Qué es async/await?" })
);
```

Módulo 5: Agentes

Tools, Structured Output, Runtime y Human-in-the-Loop

¿Qué es un Agente?

Un **agente** es un sistema que usa un LLM para decidir qué acciones tomar

❌ Sin Agentes

Flujo fijo y predecible

Usuario → LLM → Respuesta

El LLM solo genera texto

✅ Con Agentes

Flujo dinámico y adaptativo

Usuario → LLM → Tool → LLM → Respuesta

El LLM decide qué herramientas usar

Casos de Uso de Agentes

🔍 Búsqueda y análisis

Buscar información en APIs, bases de datos o web

🛠️ Automatización

Enviar emails, crear tickets, actualizar sistemas

💼 Asistentes empresariales

Consultar CRM, generar reportes, analizar datos

🤖 Agentes autónomos

Tools: Definiendo Herramientas

Tool: Función que el agente puede llamar con un esquema Zod

```
import { tool } from "@langchain/core/tools";
import { z } from "zod";

const weatherTool = tool(
  async ({ location }) => {
    // Simular llamada a API del tiempo
    return `El tiempo en ${location} es soleado, 22°C`;
  },
  {
    name: "get_weather",
    description: "Obtiene el tiempo actual de una ubicación",
    schema: z.object({
      location: z.string().describe("Ciudad o ubicación"),
    }),
  }
);
```

Tool Calling: Conectar Tools al LLM

```
import { ChatOpenAI } from "@langchain/openai";

const model = new ChatOpenAI({
  model: "gpt-4o-mini",
  temperature: 0,
});

// Vincular tools al modelo
const modelWithTools = model.bindTools([weatherTool]);

const response = await modelWithTools.invoke(
  "¿Qué tiempo hace en Barcelona?"
);

console.log(response.tool_calls);
// [{ name: "get_weather", args: { location: "Barcelona" } }]
```

💡 El LLM decide si usar la tool basándose en la descripción y el input

Ejecutando Tools

```
const response = await modelWithTools.invoke(
  "¿Qué tiempo hace en Barcelona?"
);

// El LLM decide llamar a la tool
if (response.tool_calls && response.tool_calls.length > 0) {
  const toolCall = response.tool_calls[0];

  // Ejecutar la tool manualmente
  const toolResult = await weatherTool.invoke(toolCall.args);

  console.log(toolResult);
  // "El tiempo en Barcelona es soleado, 22°C"

  // Enviar el resultado de vuelta al LLM
  const finalResponse = await modelWithTools.invoke([
    new HumanMessage("¿Qué tiempo hace en Barcelona?"),
    response,
    new ToolMessage({
      tool_call_id: toolCall.id,
      content: toolResult,
    }),
  ]);
}
```

Capacidades de los Agentes

`createAgent` proporciona capacidades avanzadas listas para usar

🔄 Loops automáticos

El agente ejecuta tools hasta completar la tarea

💾 Estado persistente

Mantiene contexto entre llamadas

👤 Human-in-the-loop

Pausar y requerir aprobación humana

Estado Persistente

Para usar estado persistente, necesitas `@langchain/langgraph`

```
npm install @langchain/langgraph
```

LangGraph proporciona los checkpointers para persistencia

Bajo el capó, `createAgent` de LangChain v1 usa LangGraph

Agente Simple con createAgent

```
import { createAgent } from "langchain";

// Crear agente con modelo y tools (v1.x)
const agent = createAgent({
  model: "openai:gpt-4o-mini",
  tools: [weatherTool],
});

// El agente maneja automáticamente:
// 1. Decidir si usar tools
// 2. Ejecutar las tools
// 3. Procesar los resultados
// 4. Repetir hasta tener respuesta final

const result = await agent.invoke({
  messages: [{ role: "user", content: "¿Qué tiempo hace en Madrid?" }]
});

console.log(result.messages[result.messages.length - 1].content);
```

✨ LangChain v1: `createAgent` reemplaza a `createReactAgent` (deprecated)

Múltiples Tools

```
import { tool } from "@langchain/core/tools";
import { z } from "zod";

const calculatorTool = tool(
  async ({ operation, a, b }) => {
    const ops = { add: a + b, subtract: a - b, multiply: a * b, divide: a / b };
    return ops[operation].toString();
  },
  {
    name: "calculator",
    description: "Realiza operaciones matemáticas básicas",
    schema: z.object({
      operation: z.enum(["add", "subtract", "multiply", "divide"]),
      a: z.number(),
      b: z.number(),
    }),
  }
);

const agent = createAgent({
  model: "openai:gpt-4o-mini",
  tools: [weatherTool, calculatorTool],  // Múltiples tools
});
```

State Management

Estado: Datos que persisten entre ejecuciones del agente

```
import { createAgent } from "langchain";
import { MemorySaver } from "@langchain/langgraph";

const agent = createAgent({
  model: "openai:gpt-4o-mini",
  tools: [weatherTool],
  store: new MemorySaver(),  // Guardar estado en memoria
});

// Primera conversación
await agent.invoke(
  { messages: [{ role: "user", content: "Hola, soy Juan" }] },
  { configurable: { thread_id: "conversation-1" } }
);

// Segunda conversación (recuerda el contexto)
await agent.invoke(
  { messages: [{ role: "user", content: "¿Cuál es mi nombre?" }] },
  { configurable: { thread_id: "conversation-1" } }
);
```

Tipos de Checkpointers

💾 MemorySaver

Guarda en memoria (RAM). Se pierde al reiniciar.

Útil para desarrollo y testing

🗄️ SqliteSaver

Guarda en SQLite (archivo local)

Persistencia local simple

Tipos de Checkpointers

🐘 PostgresSaver

Guarda en PostgreSQL

Producción, múltiples instancias

☁️ Custom Checkpointer

Implementa tu propio backend (Redis, MongoDB, etc.)

Human-in-the-Loop: Interrupciones

Interrupciones: Pausar el agente para requerir aprobación humana

```
import { createAgent } from "langchain";
import { MemorySaver } from "@langchain/langgraph";

const agent = createAgent({
  model: "openai:gpt-4o-mini",
  tools: [sendEmailTool],
  store: new MemorySaver(),
  interruptBefore: ["tools"],  // Pausar antes de ejecutar tools
});

const config = { configurable: { thread_id: "thread-1" } };

// Primera ejecución: se pausa antes de ejecutar la tool
const result1 = await agent.invoke(
  { messages: [{ role: "user", content: "Envía un email a juan@example.com" }] },
  config
);
// Estado: "interrupted" (esperando aprobación)
```

Aprobando Interrupciones

```
// Obtener el estado actual
const state = await agent.getState(config);

console.log(state.next);  // ["tools"] - siguiente paso pausado
console.log(state.values.messages);  // Ver qué tool quiere ejecutar

// Aprobar y continuar
const result2 = await agent.invoke(null, config);
// El agente continúa desde donde se pausó

// O rechazar y modificar
await agent.updateState(config, {
  messages: [{ role: "human", content: "No, no envíes ese email" }]
});

const result3 = await agent.invoke(null, config);
// El agente procesa la instrucción actualizada
```

💡 Útil para acciones sensibles: emails, pagos, eliminaciones, etc.

Streaming de Agentes

Ver los pasos del agente en tiempo real

```
const stream = await agent.stream(
  { messages: [{ role: "user", content: "¿Qué tiempo hace en Barcelona?" }] },
  { configurable: { thread_id: "thread-1" } }
);

for await (const event of stream) {
  console.log("---");
  console.log(event);
  // Muestra cada paso:
  // 1. Decisión del LLM
  // 2. Ejecución de la tool
  // 3. Resultado de la tool
  // 4. Respuesta final
}
```

Mejores Prácticas: Tools

✅ Descripciones claras

Escribe descripciones detalladas de tools para ayudar al LLM a decidir

✅ Validación con Zod

Usa esquemas Zod robustos para prevenir errores

✅ Manejo de errores

Las tools deben devolver mensajes de error útiles, no lanzar excepciones

Mejores Prácticas: Seguridad

✅ Limitar iteraciones

Configura max_iterations para evitar loops infinitos

✅ Human-in-the-loop para acciones críticas

Requiere aprobación para operaciones sensibles (emails, pagos, eliminaciones)

✅ Timeouts

Establece timeouts para tools que llaman a APIs externas

Middlewares en Agentes

Middleware: Interceptor que se ejecuta antes/después de cada paso del agente

📊 Logging y observabilidad

Registrar cada decisión y ejecución de tool

⏱️ Métricas y timing

Medir tiempo de ejecución de cada paso

🛡️ Validación y seguridad

Verificar inputs/outputs antes de continuar

Ejemplo: Middleware de Logging

```
import { createAgent } from "langchain";

// Middleware para logging
const loggingMiddleware = async (state, config, next) => {
  console.log("🔵 Antes:", state.messages.length, "mensajes");

  const result = await next(state, config);

  console.log("🟢 Después:", result.messages.length, "mensajes");

  return result;
};

const agent = createAgent({
  model: "openai:gpt-4o-mini",
  tools: [weatherTool],
  middleware: [loggingMiddleware],
});

// Cada paso del agente pasa por el middleware
await agent.invoke({
  messages: [{ role: "user", content: "¿Qué tiempo hace?" }]
});
```

Guardrails: Protección de Contenido

Guardrails: Validaciones automáticas para prevenir contenido inapropiado o peligroso

🚫 Contenido prohibido

Bloquear temas sensibles, violencia, contenido ilegal

🔒 Injection attacks

Detectar intentos de manipular el prompt del sistema

📏 Límites de uso

Controlar longitud, complejidad, tokens consumidos

PII: Información Personal Identificable

PII (Personally Identifiable Information): Datos que identifican a una persona

🔍 Ejemplos de PII

Nombres, emails, teléfonos, DNI, direcciones, números de tarjeta, IPs

⚠️ Riesgos

Los LLMs almacenan datos en sus logs y entrenamientos

Proveedores como OpenAI pueden usar los datos para mejorar sus modelos

✅ Protección

Ejemplo: Detección de PII

```
// Middleware para detectar y enmascarar PII
const piiMiddleware = async (state, config, next) => {
  const lastMessage = state.messages[state.messages.length - 1];

  // Detección simple de emails (en producción usar librerías especializadas)
  const emailRegex = /\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b/g;

  if (emailRegex.test(lastMessage.content)) {
    console.warn("⚠️ PII detectado: email");

    // Opción 1: Bloquear
    throw new Error("No se permite enviar emails al LLM");

    // Opción 2: Enmascarar
    const masked = lastMessage.content.replace(emailRegex, "[EMAIL]");
    state.messages[state.messages.length - 1].content = masked;
  }

  return await next(state, config);
};

const agent = createAgent({
  model: "openai:gpt-4o-mini",
  tools: [],
```

Librerías para PII y Guardrails

🛡️ NeMo Guardrails (NVIDIA)

Framework completo para guardrails en LLMs

github.com/NVIDIA/NeMo-Guardrails

🔍 Microsoft Presidio

Detección y anonimización de PII

github.com/microsoft/presidio

🚦 Guardrails AI

Validación de outputs con reglas personalizadas

Módulo 6: LangSmith

Observabilidad, debugging y evaluación

¿Qué es LangSmith?

LangSmith: Plataforma de observabilidad y testing para aplicaciones LLM

🔍 Observabilidad

Ver cada paso del agente: prompts, respuestas, tools ejecutadas

🐛 Debugging

Identificar errores, latencias y costos en producción

✅ Evaluación

Datasets de prueba y métricas de calidad

¿Por qué LangSmith?

❌ Sin LangSmith

• No sabes qué prompts se enviaron exactamente

• Difícil reproducir errores

• Imposible saber el costo real por usuario

• Testing manual y tedioso

✅ Con LangSmith

• Trazas completas de cada ejecución

• Replay de conversaciones problemáticas

• Dashboard de costos y latencias

Configuración de LangSmith

Paso 1: Crear cuenta gratuita en smith.langchain.com

Paso 2: Obtener API Key desde Settings → API Keys

Paso 3: Configurar variables de entorno

```
# .env
LANGCHAIN_TRACING_V2=true
LANGCHAIN_API_KEY=lsv2_pt_...
LANGCHAIN_PROJECT=mi-proyecto
```

✨ Automático: LangChain detecta las variables y envía trazas a LangSmith

Ejemplo con LangSmith

```
import { createAgent } from "langchain";

// Con las variables de entorno configuradas,
// las trazas se envían automáticamente a LangSmith

const agent = createAgent({
  model: "openai:gpt-4o-mini",
  tools: [weatherTool],
});

const result = await agent.invoke({
  messages: [{ role: "user", content: "¿Qué tiempo hace en Madrid?" }]
});

// ✅ Ve la traza completa en https://smith.langchain.com
// - Input del usuario
// - Decisión del LLM de usar la tool
// - Ejecución de la tool
// - Respuesta final
// - Tokens consumidos
// - Latencia de cada paso
```

Plan Gratuito

🎁 Developer (Gratis)

- 5,000 traces/mes
- Retención de 14 días
- 1 usuario
- Perfecto para desarrollo y testing

📚 Planes de pago desde $39/mes (Plus) para equipos y producción

smith.langchain.com/pricing

Próximos Pasos

🚀 En producción

Activa LangSmith para monitorear tu aplicación en real-time

📚 Explora datasets

Crea casos de prueba y evalúa mejoras en tus prompts

📖 Documentación

docs.smith.langchain.com

Eskerrik asko! 🎉

Gracias por vuestra atención

¿Dudas o consultas?

📧 **jabi.infante@gmail.com**