



UNIVERSIDAD DE CHILE

Inteligencia Artificial Generativa

Let's talk about hype stuff

Valentin Barriere // Clemente Henriquez

Universidad de Chile – DCC

Diplomado de Postítulo en Inteligencia Artificial, Primavera 2025

Advanced LLM

Outline : El Viaje hasta Aquí: De Transformers a Agentes

El Viaje hasta Aquí: De
Transformers a Agentes

¿Qué es un Agente?

Function Calling y Tool Use

Model Context Protocol (MCP)

Sistemas Multi-Agente

Fronteras de Investigación en
LLMs

Resumen: El Camino Recorrido en el Módulo

¿Dónde comenzamos?

Este módulo inició con la arquitectura fundamental que revolucionó el procesamiento del lenguaje: el **Transformer** (Vaswani et al., 2017).

La evolución que hemos recorrido:

1. **Transformers:** Arquitectura base con self-attention
 - Mecanismo de atención que captura dependencias de largo alcance
 - Paralelización eficiente del entrenamiento
 - Base para todos los LLMs modernos
2. **Large Language Models (LLMs):** Escalar Transformers
 - GPT, BERT, T5: primeros modelos de gran escala
 - Emergencia de capacidades con el tamaño
 - Modelos generativos que generan texto coherente

Resumen: Haciendo los LLMs Más Útiles

3. RAG (Retrieval-Augmented Generation): Añadir conocimiento externo

- Problema: LLMs tienen conocimiento estático y limitado
- Solución: Recuperar información relevante de bases de datos
- IR clásico (BM25) + Embeddings densos (neural retrieval)
- Arquitectura: Retrieve → Augment → Generate

4. Multimodalidad: Más allá del texto

- Integrar visión, audio y texto
- Vision Transformers (ViT), CLIP, Flamingo
- Modelos que entienden y generan múltiples modalidades
- GPT-4V, Gemini, Claude 3.5: texto + imágenes

El objetivo siempre fue el mismo

Hacer que los LLMs sean **más útiles**, **más precisos**, y capaces de resolver **tareas reales**.

El Problema Fundamental: Limitaciones de un LLM Estático

A pesar de todas las mejoras, los LLMs por sí solos tienen limitaciones fundamentales:

1. Conocimiento congelado

- Datos de entrenamiento quedan obsoletos
- No pueden acceder a información privada o actualizada
- RAG ayuda, pero es limitado

2. Solo generan texto

- No pueden ejecutar código
- No pueden hacer cálculos matemáticos precisos
- No pueden interactuar con APIs o sistemas externos

3. Sin memoria persistente

- Cada sesión es independiente
- Olvidan contexto entre conversaciones
- Limitados por ventana de contexto (aunque cada vez más grande)

4. Respuestas únicas, sin iteración

- Un prompt → una respuesta
- No pueden validar, corregir errores, o refinar
- No hay planificación multi-step

La Evolución Natural: El Agente

¿Qué necesitamos para resolver tareas complejas del mundo real?

Tareas complejas requieren:

- **Razonamiento multi-step:** Descomponer problemas en sub-tareas
- **Acceso a herramientas:** Calculadora, búsqueda web, APIs, bases de datos
- **Planificación:** Decidir qué hacer en cada paso
- **Ejecución:** Tomar acciones en el entorno
- **Validación y corrección:** Detectar errores y re-intentar
- **Memoria:** Recordar interacciones previas y aprender de ellas

La solución: Agentes Autónomos

Un **agente** es un sistema que combina un LLM con capacidades adicionales para **razonar**, **planificar**, **usar herramientas** y **actuar** de forma autónoma e iterativa.

Outline : ¿Qué es un Agente?

El Viaje hasta Aquí: De
Transformers a Agentes

¿Qué es un Agente?

Function Calling y Tool Use

Model Context Protocol (MCP)

Sistemas Multi-Agente

Fronteras de Investigación en
LLMs

Definición Formal de Agente

Definición

Un **agente** es un sistema autónomo que combina un LLM con componentes adicionales para percibir su entorno, razonar sobre él, y tomar acciones para alcanzar objetivos.

Formulación matemática:

$$\text{Agent} = \text{LLM} + \text{Memory} + \text{Tools} + \text{Planning} + \text{Action Loop}$$

Componentes fundamentales:

1. **LLM Core:** El "cerebro" que razona y decide
2. **Memory System:** Almacenamiento de contexto y experiencias previas
3. **Tools/Actions:** Funciones externas que el agente puede ejecutar
4. **Planning Module:** Capacidad de descomponer tareas y crear planes
5. **Action Loop:** Ciclo iterativo de observación, pensamiento y acción

El Ciclo Básico de un Agente

El paradigma central de un agente es el loop de acción:

Observe → Think → Plan → Act →
Observe → ...

Desglose del ciclo:

1. **Observe (Percibir):** Recibir información del entorno
 - Input del usuario, resultados de acciones previas, estado del mundo
2. **Think (Razonar):** Analizar la situación actual
 - "¿Qué información tengo?", "¿Qué me falta?", "¿Entiendo el problema?"
3. **Plan (Planificar):** Decidir qué hacer
 - "¿Cuál es el próximo paso más útil?", "¿Qué herramienta necesito usar?"
4. **Act (Actuar):** Ejecutar una acción
 - Llamar a una herramienta, generar respuesta, modificar el entorno

Contraste: LLM vs Agente

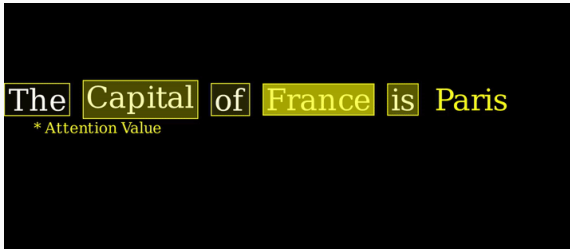


Figure 1: Ilustración Interacción Básica con un LLM. Fuente: [1]

Contraste: LLM vs Agente



Figure 2: Ilustración Interacción con un Agente. Fuente: [2]

Ejemplo Concreto: LLM vs Agente

Tarea: "¿Cuántas personas viven en la ciudad natal del actual presidente de Chile?"

LLM sin herramientas:

LLM: "Gabriel Boric, el presidente actual de Chile (hasta mi conocimiento de enero 2023), nació en Punta Arenas. La población de Punta Arenas es de aproximadamente 130,000 habitantes."

Problemas:

- Conocimiento desactualizado (puede haber cambiado)
- Cifra imprecisa ("aproximadamente")
- No puede verificar información

Ejemplo Concreto: LLM vs Agente

Agente con herramientas:

Thought: Necesito saber quién es el presidente actual de Chile.

Action: `search_web("presidente actual Chile 2024")`

Observation: Gabriel Boric es el presidente de Chile desde 2022.

Thought: Ahora necesito saber su ciudad natal.

Action: `search_web("Gabriel Boric ciudad natal")`

Observation: Gabriel Boric nació en Punta Arenas, Chile.

Thought: Finalmente necesito la población actualizada de Punta Arenas.

Action: `search_web("población Punta Arenas Chile 2024")`

Observation: Según censo 2024, Punta Arenas tiene 127,369 habitantes.

Thought: Ya tengo toda la información verificada y actualizada.

Answer: Aproximadamente 127,369 personas viven en Punta Arenas,
la ciudad natal de Gabriel Boric.

¿Por Qué Agentes? Motivación con Ejemplos

1. Investigación multi-fuente:

- "Compara las políticas climáticas de 5 países europeos"
- Requiere: búsqueda web \times 5, extracción de datos, síntesis

2. Análisis de datos con código:

- "Analiza este CSV y encuentra patrones anómalos"
- Requiere: leer archivo, escribir código, ejecutar, interpretar resultados, refinar

3. Workflows de múltiples pasos:

- "Busca un restaurante italiano, reserva mesa, y envía calendario"
- Requiere: búsqueda, API de reservas, API de calendario

4. Tareas con validación y retry:

- "Escribe tests para este código y asegúrate de que pasen"
- Requiere: generar tests, ejecutar, si fallan \rightarrow corregir \rightarrow re-ejecutar

Clasificación según arquitectura y nivel de autonomía:

1. Simple Reflex Agents (Reactivos simples)

- Responden directamente a input sin memoria
- Ejemplo: Chatbot básico que responde preguntas
- Limitación: No pueden manejar secuencias de acciones

Clasificación según arquitectura y nivel de autonomía:

2. Model-Based Reflex Agents (Con estado interno)

- Mantienen representación interna del estado
- Ejemplo: Agente que recuerda conversaciones previas
- Mejor que simple reflex, pero planning limitado

Clasificación según arquitectura y nivel de autonomía:

3. Goal-Based Agents (Basados en objetivos)

- Planifican secuencias de acciones para alcanzar un goal
- Ejemplo: Agente de investigación que genera plan y lo ejecuta
- Usan búsqueda y planning para decidir acciones

Clasificación según arquitectura y nivel de autonomía:

4. Utility-Based Agents (Basados en utilidad)

- Optimizan una función de utilidad/recompensa
- Ejemplo: Trading bot que maximiza ganancias
- Más sofisticados, requieren definir métricas de éxito

En el contexto de LLMs, un agente típico combina:

Stack tecnológico común:

1. **LLM Core:** GPT-4, Claude 3.5, Llama 3.1, Gemini
2. **Prompting Strategy:** ReAct, Chain-of-Thought, Tree-of-Thoughts
3. **Tool Registry:** Funciones disponibles con schemas JSON
4. **Memory Store:**
 - Short-term: ventana de contexto del LLM
 - Long-term: vector database (Chroma, Pinecone, Weaviate)
5. **Orchestration Layer:** LangChain, LangGraph, AutoGen, CrewAI
6. **Execution Environment:** Sandboxed code execution, API calls

Todo esto lo veremos en detalle en las próximas secciones.

Agentes vs Sistemas Tradicionales

Aspecto	Sistema Tradicional	Agente LLM
Programación	Explícita (if/else)	Implícita (prompts + tools)
Adaptabilidad	Requiere re-programar	Se adapta via prompts
Manejo de ambigüedad	Falla o requiere casos específicos	Maneja lenguaje natural
Planning	Hard-coded workflows	Genera planes dinámicamente
Error handling	Try-catch explícitos	Puede razonar sobre errores
Extensibilidad	Agregar código nuevo	Agregar tools con descripción

Ejemplo comparativo

Tarea: "Busca hoteles en París con buenas reseñas y precio moderado"

Sistema tradicional: Requiere API calls explícitas, parsing, filtros hard-coded

Agente: Usa tool de búsqueda, interpreta "buenas reseñas" (¿4 estrellas), "precio moderado" (contexto del usuario), y presenta resultados

Ventaja clave: Flexibilidad y capacidad de manejar tareas complejas sin programación explícita.

El Futuro es Agentic: ¿Por Qué Ahora?

¿Por qué los agentes están despegando justo ahora en 2024-2025?

Factores convergentes:

1. LLMs suficientemente capaces

- Razonamiento robusto (GPT-4, Claude 3.5, Gemini)
- Function calling nativo
- Contextos largos (100k+ tokens)

2. Frameworks maduros

- LangChain, LlamaIndex, AutoGen simplificaron desarrollo
- Herramientas de observabilidad (LangSmith, Weights & Biases)

3. Infraestructura disponible

- APIs rápidas y accesibles
- Vector databases eficientes
- Code execution environments (Jupyter kernels, sandboxes)

4. Casos de uso demostrables

- ChatGPT Code Interpreter fue un "aha moment"
- Devin (software engineering agent) generó expectación
- Research agents (Perplexity, Elicit) muestran valor

Transición a la Siguiente Sección

Hemos establecido:

- El viaje desde Transformers hasta Agentes
- Qué es un agente y por qué lo necesitamos
- Cómo un agente difiere de un LLM estático
- La historia reciente y el momentum actual

Lo que viene

Ahora profundizaremos en la **tecnología fundamental** que hace posibles los agentes: **Function Calling y Tool Use**.

Veremos:

- Cómo los LLMs aprenden a llamar funciones
- Arquitectura técnica del function calling
- Implementación práctica con HuggingFace Transformers
- Model Context Protocol (MCP): el futuro estándar

Outline : Function Calling y Tool Use

El Viaje hasta Aquí: De
Transformers a Agentes

¿Qué es un Agente?

Function Calling y Tool Use

Model Context Protocol (MCP)

Sistemas Multi-Agente

Fronteras de Investigación en
LLMs

¿Qué es Function Calling?

- El LLM puede "llamar" funciones externas de forma **estructurada**
- El modelo predice: `nombre_función + argumentos_JSON`
- El sistema ejecuta la función y devuelve el resultado al modelo
- El modelo usa el resultado para continuar su razonamiento

Ejemplo de flujo con el clima en Santiago:

1. Usuario hace pregunta

User: "¿Cuál es el clima en Santiago?"

2. LLM decide usar función y genera JSON estructurado

LLM genera:

```
{  
  "function": "get_weather",  
  "arguments": {  
    "location": "Santiago, Chile",  
    "unit": "celsius"  
  }  
}
```

3. Sistema ejecuta la función get_weather("Santiago, Chile", "celsius")

Sistema ejecuta  Devuelve: {"temp": 18, "condition": "sunny", "humidity": 45}

4. LLM recibe resultado y genera respuesta final para el usuario

LLM genera respuesta final:

"En Santiago hace 18°C, está soleado y la humedad es del 45%."

Prerequisito: Definir las Funciones Disponibles

Antes de usar function calling, debemos definir qué funciones están disponibles.

El concepto fundamental

Function calling NO significa que el LLM ejecuta código arbitrario. El LLM solo **predice** qué función llamar y con qué argumentos. Nosotros ejecutamos la función.

El flujo completo:

1. **Definimos** un set de funciones disponibles (con schemas JSON)
2. **Pasamos** estas definiciones al LLM en el system prompt
3. **El LLM predice** qué función llamar (formato JSON estructurado)
4. **Nosotros ejecutamos** la función en nuestro código
5. **Devolvemos** el resultado al LLM
6. **El LLM continúa** su razonamiento con el resultado

1. Fine-tuning especializado:

- Entrenamiento en miles de ejemplos de llamadas a funciones
- Datasets sintéticos: (query, función correcta, argumentos)
- Ejemplo: "¿Qué hora es en Tokyo?" →
`get_time(location="Tokyo")`
- Los modelos aprenden el patrón de cuándo y cómo llamar funciones

2. System prompt con schema de herramientas:

- Definición de todas las tools disponibles en el contexto
- Cada tool tiene: nombre, descripción, parámetros con tipos
- El LLM consulta este "catálogo" antes de decidir qué usar

3. Tokens especiales:

- Delimitadores para marcar inicio/fin de function calls
- Ejemplo: `<tool_call>`, `</tool_call>`, `<tool_response>`
- Permiten parsing confiable del output del modelo

Arquitectura Técnica del Function Calling (2/2)

Schema de herramientas (formato OpenAI/HuggingFace):

El schema define la "firma" de cada función disponible:

```
{
  "name": "get_weather",
  "description": "Obtiene el clima actual de una ubicación específica.  
Útil cuando el usuario pregunta por el tiempo o clima.",
  "parameters": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "Ciudad y país en formato 'Ciudad, País'"
      },
      "unit": {
        "type": "string",
        "enum": ["celsius", "fahrenheit"],
        "description": "Unidad de temperatura a usar"
      }
    },
    "required": ["location"]
  }
}
```

Componentes críticos del schema

- **name:** Identificador único de la función
- **description:** Crucial - el LLM decide usarla basado en esto
- **parameters:** Tipos, descripciones, restricciones (enum, required)

Modelos con Soporte Nativo de Function Calling

Modelos Proprietarios

- **OpenAI:** GPT-4, GPT-4o, GPT-4-turbo, GPT-3.5-turbo
- **Anthropic:** Claude 3.5 Opus, Sonnet, Haiku
- **Google:** Gemini 1.5 Pro / Flash, Gemini 2.0 Flash

Modelos Open Source

- **Meta Llama:** Llama 3.1 (8B–405B), Llama 3.3 70B
- **Mistral:** Mistral Large / Small, Mixtral 8x7B y 8x22B
- **Qwen:** Qwen 2.5 (7B–72B), Qwen 2.5 Coder
- **Google:** Gemma 2 (9B, 27B)

¿Cómo verificar si un modelo soporta tools?

1. **Documentación oficial:** Revisar model card
2. **Tokenizer:** Ver si `chat_template` incluye lógica para `tools`
3. **Frameworks:** Probar con `transformers` o `langchain`

Implementación en HuggingFace Transformers (1/2)

Ejemplo completo con Llama 3.1:

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import json

# 1. Cargar modelo y tokenizer
model_id = "meta-llama/Llama-3.1-8B-Instruct" # Versión más liviana
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    device_map="auto",
    torch_dtype="auto"
)

# 2. Definir herramientas disponibles
tools = [
    {
        "type": "function",
        "function": {
            "name": "calculate",
            "description": "Evalúa una expresión matemática. Útil para cálculos precisos.",
            "parameters": {
                "type": "object",
                "properties": {
                    "expression": {
                        "type": "string",
                        "description": "Expresión matemática a evaluar (ej: '234 * 567')"
                    }
                }
            },
            "required": ["expression"]
        }
    }
]
```

Implementación en HuggingFace Transformers (2/2)

```
# 3. Crear mensajes con pregunta del usuario
messages = [
    {"role": "user", "content": "¿Cuánto es 234 multiplicado por 567?"}
]

# 4. Apply chat template WITH tools
# Esto formatea el prompt con las definiciones de tools
inputs = tokenizer.apply_chat_template(
    messages,
    tools=tools, # ¡Clave! Pasar las tools aquí
    add_generation_prompt=True,
    return_tensors="pt"
).to(model.device)

# 5. Generar respuesta
outputs = model.generate(inputs, max_new_tokens=256)
response = tokenizer.decode(outputs[0], skip_special_tokens=False)

print(response)
# Output esperado:
# <tool_call>{"name": "calculate", "arguments": {"expression": "234 * 567"}}</tool_call>

# 6. Parsear la tool call
tool_call = parse_tool_call(response) # Extraer JSON
result = eval(tool_call["arguments"]["expression"]) # Ejecutar: 132678

# 7. Enviar resultado de vuelta al modelo
messages.append({"role": "assistant", "tool_calls": [tool_call]})
messages.append({"role": "tool", "content": str(result)})

# 8. Generar respuesta final
final_response = generate_with_messages(model, tokenizer, messages)
print(final_response)
# Output: "El resultado de 234 multiplicado por 567 es 132,678."
```

Paso Clave: Implementar las Funciones Reales

El LLM solo predice qué función llamar. NOSOTROS ejecutamos la función.

```
# Implementar las funciones disponibles en Python
def calculate(expression: str) -> str:
    """
    Evalúa una expresión matemática de forma segura.
    Args:
        expression: Expresión a evaluar (ej: "234 * 567")
    Returns:
        str: Resultado del cálculo
    """
    try:
        # Opción 1: eval() - PELIGROSO en producción, solo para demos
        # result = eval(expression)

        # Opción 2: Usar ast.literal_eval para expresiones simples
        # Opción 3: Usar una librería como numexpr o simpleeval (más seguro)
        import ast
        import operator

        # Parsear y evaluar de forma segura
        # Para este ejemplo simplificado usamos eval con cuidado
        allowed_chars = set('0123456789+-*/() .')
        if not all(c in allowed_chars for c in expression):
            return "Error: Expresión contiene caracteres no permitidos"

        result = eval(expression)
        return str(result)
    except Exception as e:
        return f"Error al calcular: {str(e)}"
```

Paso Clave: Implementar las Funciones Reales

```
def get_weather(location: str, unit: str = "celsius") -> str:
    """
    Obtiene el clima de una ubicación.
    En producción, esto llamaría a una API real como OpenWeatherMap.
    """
    # Simulación para el ejemplo
    fake_data = {
        "Santiago, Chile": {"temp": 18, "condition": "sunny"},
        "Tokyo, Japan": {"temp": 12, "condition": "cloudy"}
    }
    data = fake_data.get(location, {"temp": 20, "condition": "unknown"})
    return json.dumps(data)
```

Parsing y Ejecución: El Ciclo Completo

1. Parsear la tool call del modelo:

```
import re
import json

def parse_tool_call(response: str) -> dict:
    """Extrae el JSON de la tool call del response del modelo"""
    pattern = r'<tool_call>(.*?)</tool_call>'
    match = re.search(pattern, response, re.DOTALL)
    if match:
        tool_call_json = match.group(1).strip()
        return json.loads(tool_call_json)
    return None

# Ejemplo de uso
response = """<tool_call>{"name": "calculate", "arguments": {"expression": "234 *
↪ 567"}}</tool_call>"""
tool_call = parse_tool_call(response)
print(tool_call) # {"name": "calculate", "arguments": {"expression": "234 * 567"}}
```

2. Ejecutar la función correspondiente:

```
# Mapeo de nombres a funciones reales
AVAILABLE_FUNCTIONS = {
    "calculate": calculate,
    "get_weather": get_weather
}

# Ejecutar la función
function_name = tool_call["name"]
function_args = tool_call["arguments"]

if function_name in AVAILABLE_FUNCTIONS:
    function_to_call = AVAILABLE_FUNCTIONS[function_name]
    result = function_to_call(**function_args) # Ejecutar con argumentos
    print(f"Resultado: {result}") # "132678"
else:
    print(f"Error: Función {function_name} no existe")
```

Multiple Tools: Ejemplo más Completo

Definir múltiples herramientas para un agente más capaz:

```
tools = [  
  {  
    "type": "function",  
    "function": {  
      "name": "search_web",  
      "description": "Busca información en internet. Útil para preguntas sobre hechos  
↔ actuales.",  
      "parameters": {  
        "type": "object",  
        "properties": {  
          "query": {"type": "string", "description": "Término de búsqueda"}  
        },  
        "required": ["query"]  
      }  
    }  
  },  
  {  
    "type": "function",  
    "function": {  
      "name": "calculate",  
      "description": "Calcula expresiones matemáticas con precisión.",  
      "parameters": {  
        "type": "object",  
        "properties": {  
          "expression": {"type": "string"}  
        },  
        "required": ["expression"]  
      }  
    }  
  },  
  {  
    "type": "function",  
    "function": {  
      "name": "get_weather",  
      "description": "Obtiene el clima de una ciudad.",  
      "parameters": {  
        "type": "object",  
        "properties": {  
          "city": {"type": "string", "description": "Nombre de la ciudad"}  
        },  
        "required": ["city"]  
      }  
    }  
  }  
]
```

Outline : Model Context Protocol (MCP)

El Viaje hasta Aquí: De
Transformers a Agentes

¿Qué es un Agente?

Function Calling y Tool Use

Model Context Protocol (MCP)

Sistemas Multi-Agente

Fronteras de Investigación en
LLMs

MCP: La Analogía de los Enchufes

Imaginemos los años 90s: Cada dispositivo electrónico tenía su propio conector propietario.

Antes de USB:

- Cada celular: cargador diferente
- Cada cámara: cable único
- Cada impresora: conector propio
- Resultado: Cajón lleno de cables incompatibles

Era un caos:

- No intercambiable
- Difícil de reemplazar
- Costoso mantener

Después de USB:

- Un solo tipo de conector
- Funciona con cualquier dispositivo
- Estandarizado globalmente
- Ecosistema masivo de accesorios

Beneficio enorme:

- Plug & play universal
- Reutilización
- Innovación acelerada

El Problema que MCP Resuelve

Situación actual con Function Calling tradicional:

- **Cada proveedor tiene su formato:**
 - OpenAI usa un formato JSON específico
 - Anthropic usa otro formato ligeramente diferente
 - Cada modelo open source puede tener su propia convención
- **Cada aplicación reinventa la rueda:**
 - Si quieres que tu app use PostgreSQL, escribes código custom
 - Si quieres agregar Google Drive, más código custom
 - Si cambias de LLM provider, reescribes todo
- **No hay reutilización:**
 - Una herramienta para app A no sirve para app B
 - Duplicación masiva de esfuerzo
 - Difícil de mantener y actualizar

La visión de MCP

Un protocolo universal donde escribes un "servidor" de herramientas una vez, y funciona con cualquier aplicación que soporte MCP.

¿Qué es MCP?

Definición

Model Context Protocol (MCP) es un protocolo abierto desarrollado por Anthropic (noviembre 2024) que estandariza cómo los LLMs acceden a herramientas y datos externos.

Idea central:

1. Defines un **MCP Server** que expone herramientas (ej: "servidor de PostgreSQL")
2. Cualquier aplicación con un **MCP Client** puede conectarse a ese servidor
3. El servidor funciona con Claude, ChatGPT, VS Code, o cualquier app compatible
4. No necesitas reescribir nada si cambias de LLM o aplicación

Estado actual:

- Protocolo en desarrollo activo con SDKs disponibles en Python y TypeScript
- Adopción creciente, con comunidades creando servers para casos comunes

Arquitectura de MCP: Componentes

El ecosistema MCP tiene 3 componentes principales:

1. **MCP Host** (Tu aplicación con LLM)

- Ejemplos: Claude Desktop, VS Code, tu propia app
- Es quien "usa" las herramientas
- Decide qué servers conectar

2. **MCP Server** (Provee las herramientas)

- Expone tools, datos, prompts
- Ejemplos: servidor de PostgreSQL, servidor de Slack
- Puede conectarse a múltiples hosts simultáneamente

3. **Data Source** (La fuente real de datos)

- Base de datos, API, filesystem, etc.
- El server se conecta a ella
- Puede ser local o remota

Flujo: Host ↔ Server ↔ Data Source

¿Qué Puede Hacer un MCP Server?

Un MCP Server puede exponer:

1. Tools (Herramientas) - Lo más común

- Acciones que el LLM puede ejecutar
- Ejemplos: queries a DB, buscar archivos, enviar emails
- Es el equivalente a function calling, pero estandarizado

2. Resources (Recursos) - Para lectura

- Datos que el LLM puede leer
- Ejemplos: contenido de archivos, registros de DB
- El server expone URIs que el host puede consultar

3. Prompts (Plantillas) - Workflows predefinidos

- Templates reutilizables con contexto
- Ejemplos: "Analiza este código", "Resume este documento"

Ejemplo: MCP Server Completo

```
from mcp.server import Server
from mcp.types import Tool, TextContent

# 1. Crear el server
server = Server("calculator-server")

# 2. Definir herramientas disponibles
@server.list_tools()
async def list_tools() -> list[Tool]:
    return [Tool(name="calculate", description="Calcula expresiones matemáticas",
        inputSchema={"type": "object", "properties":
            {"expr": {"type": "string"}}, "required": ["expr"]})]

# 3. Implementar la lógica
@server.call_tool()
async def call_tool(name: str, args: dict) -> list[TextContent]:
    if name == "calculate":
        result = eval(args["expr"]) # Evaluar expresión
        return [TextContent(type="text", text=str(result))]
```

Eso es todo. El server ya puede ser usado por cualquier MCP host.

Para usarlo:

```
python calculator_server.py
```

Conectar un Server a tu App (Host)

Configuración simple en Claude Desktop:

```
// claude_desktop_config.json
{
  "mcpServers": {
    "calculator": {
      "command": "python",
      "args": ["calculator_server.py"]
    }
  }
}
```

Eso es todo. Ahora Claude tiene acceso a esa calculadora.

En uso:

- User: "¿Cuánto es $234 * 567$?"
- Claude ve el tool "calculate" disponible
- Llama: `calculate(expr="234 * 567")`
- Recibe: "132678"
- Responde: "El resultado es 132,678"

Ecosistema: Servers Ya Disponibles

La comunidad ya creó servers para casos comunes:

Bases de datos:

- postgresql
- sqlite
- mysql

Desarrollo:

- git
- github
- filesystem

Repo oficial:

<https://github.com/modelcontextprotocol/servers>

Productividad:

- google-drive
- slack
- gmail

Automatización:

- puppeteer
- playwright

¿Por Qué MCP Importa?

Ventajas del protocolo estandarizado:

1. Reutilización total

- Escribe un server una vez
- Funciona con Claude, ChatGPT, Cursor, VS Code, tu propia app...
- No más código custom para cada provider

2. Composición fácil

- Combina múltiples servers: PostgreSQL + Slack + GitHub
- Solo agregar líneas en el config
- Cada server es independiente y modular

3. Ecosistema compartido

- Protocolo abierto, sin vendor lock-in
- Comunidad contribuye servers
- Beneficio colectivo (como npm, pip, etc.)

Function Calling vs MCP: ¿Cuál Usar?

Function Calling Clásico

Ventajas:

- Más simple para comenzar
- No requiere servidor externo
- Menos overhead inicial

Desventajas:

- Cada provider es diferente
- No reutilizable
- Mantenimiento difícil a escala

MCP (Protocolo)

Ventajas:

- Estándar universal
- Reutilización total
- Ecosistema compartido

Desventajas:

- Más complejo inicialmente
- Requiere servidor separado
- Protocolo aún joven

Conclusión

Ambos coexistirán. Function calling para simplicidad, MCP para escala y estandarización.

Hemos cubierto:

- Cómo funciona function calling a nivel técnico
- Implementación práctica con HuggingFace Transformers
- Qué modelos soportan function calling nativamente
- Model Context Protocol: el futuro estándar
- Cómo crear y usar MCP servers

Outline : Sistemas Multi-Agente

El Viaje hasta Aquí: De
Transformers a Agentes

¿Qué es un Agente?

Function Calling y Tool Use

Model Context Protocol (MCP)

Sistemas Multi-Agente

Fronteras de Investigación en
LLMs

¿Por Qué Multi-Agentes?

Limitaciones de un agente único:

- **Complejidad limitada:** Un agente solo puede manejar tareas hasta cierto nivel
- **Jack of all trades, master of none:** Difícil ser experto en todo
- **Escalabilidad:** Tareas muy complejas requieren demasiados steps
- **Sin verificación cruzada:** No hay quien critique o valide las decisiones

La solución: División del trabajo

Similar a equipos humanos, crear **agentes especializados** que colaboran. Cada agente es experto en su dominio y se enfoca en su área.

Patrones de Comunicación Multi-Agente

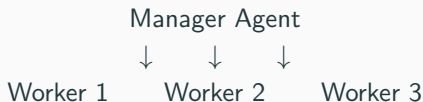
¿Cómo coordinan los agentes? Hay varios patrones:

1. Sequential (Pipeline):

Agent A \rightarrow Agent B \rightarrow Agent C \rightarrow Result

- Output de uno es input del siguiente
- Ejemplo: Researcher \rightarrow Analyst \rightarrow Writer

2. Hierarchical (Manager-Worker):



- Manager delega sub-tareas y coordina resultados
- Workers se enfocan en tareas específicas

3. Collaborative (Peer-to-peer):

Agent A \leftrightarrow Agent B \leftrightarrow Agent C

- Agentes debaten, critican y refinan juntos
- No hay jerarquía, todos contribuyen en igualdad

Caso Real: Sistema de Investigación de Anthropic

Anthropic publicó un sistema multi-agente para research profundo.

El desafío

Investigar un tema complejo requiere: búsqueda exhaustiva, lectura de múltiples fuentes, síntesis crítica, verificación cruzada de hechos, y redacción coherente. Demasiado para un solo agente.

Su solución: 4 agentes especializados trabajando juntos

1. Research Coordinator (Manager)

- Descompone el tema en sub-preguntas
- Asigna tareas a workers
- Integra resultados finales

2. Web Searcher (Worker especializado)

- Busca información en internet
- Filtra fuentes confiables vs no confiables
- Extrae contenido relevante

Sistema de Investigación de Anthropic (continuación)

3. **Content Analyzer** (Worker especializado)

- Lee y analiza documentos largos
- Extrae facts clave y citas
- Identifica contradicciones entre fuentes

4. **Report Writer** (Worker especializado)

- Sintetiza información en reporte coherente
- Estructura lógica con secciones
- Cita fuentes apropiadamente

El flujo completo:

1. Coordinator recibe query: "Analiza el impacto de IA en educación"
2. Coordinator genera sub-preguntas: "¿Cómo se usa IA en K-12?", "¿Qué dicen los estudios sobre efectividad?", etc.
3. Web Searcher busca en paralelo para cada sub-pregunta
4. Content Analyzer procesa los documentos encontrados
5. Report Writer genera secciones del reporte
6. Coordinator integra todo y valida coherencia

Componentes técnicos clave:

1. Shared Context Store

- Todos los agentes leen/escriben a un contexto compartido
- Evita re-búsqueda de información ya encontrada
- Vector database para retrieval eficiente

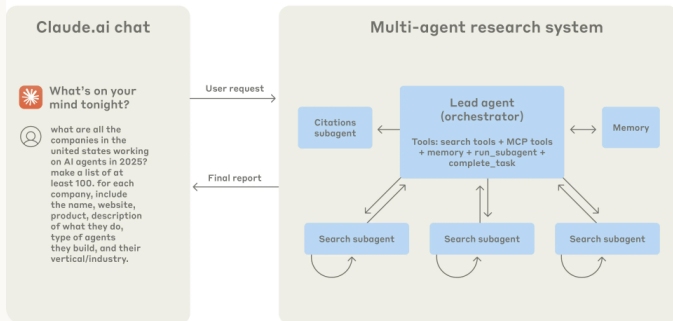
2. Task Queue

- Coordinator pone tareas en la cola
- Workers toman tareas disponibles
- Permite paralelización natural

3. Quality Checker

- Agente adicional que valida outputs
- Verifica que citas sean correctas
- Detecta alucinaciones y pide corrección

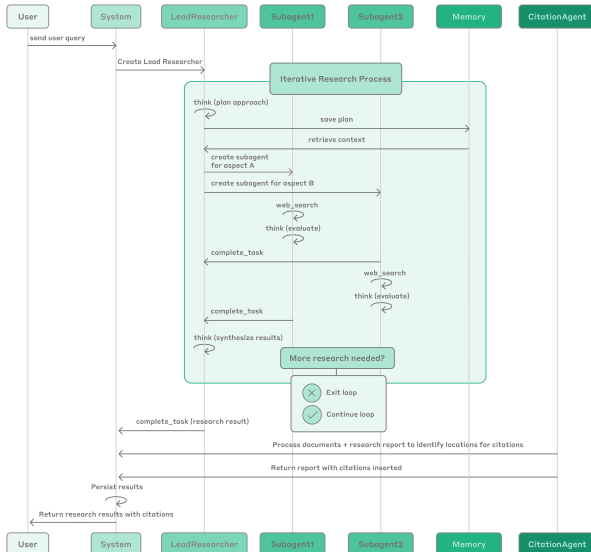
High-level Architecture of Advanced Research



The multi-agent architecture in action: user queries flow through a lead agent that creates specialized subagents to search for different aspects in parallel.

Figure 3: Caption

Multi-agent System Process Diagram



Principios para Agentes Efectivos (1/2)

- **Comprender al agente:** simular pasos revela errores (herramientas mal elegidas, búsquedas redundantes, continuar innecesariamente).
- **Delegación clara:** cada subagente necesita objetivo, formato, límites y herramientas definidas para evitar duplicación.
- **Escalar esfuerzo:** reglas para asignar recursos
 - Tareas simples → 1 agente
 - Comparaciones → 2–4 agentes
 - Investigación profunda → 10+ agentes

Principios para Agentes Efectivos (2/2)

- **Diseño de herramientas:** descripciones claras; elegir mal la herramienta provoca fallos estructurales.
- **Agentes que optimizan herramientas:** modelos pueden detectar fallas y reescribir prompts/descripciones (−40% tiempo).
- **Búsqueda eficaz:** empezar amplio, luego refinar para evitar consultas demasiado específicas.
- **Pensamiento guiado:** “extended thinking” mejora planificación y uso de herramientas.
- **Paralelización:** subagentes y herramientas en paralelo aceleran investigaciones hasta 90%.

La industria convergió en varios frameworks populares:

1. AutoGen (Microsoft, 2023-2024)

- Framework conversacional multi-agente
- Agentes se comunican via mensajes
- Soporta code execution, browsing, tool use
- Usado en producción por equipos de Microsoft

2. CrewAI (2024)

- Enfoque en roles bien definidos
- Cada agente tiene: role, goal, backstory, tools
- Workflows declarativos fáciles de configurar
- Muy popular en startups y proyectos nuevos

3. LangGraph (LangChain, 2024)

- Basado en grafos con estado

Definir un equipo de 3 agentes con CrewAI:

```
from crewai import Agent, Task, Crew

# Agente 1: Investigador
researcher = Agent(
    role='Investigador Senior',
    goal='Encontrar información actualizada y confiable sobre {topic}',
    backstory='Experto en research académico con 10 años de experiencia',
    tools=[web_search_tool, arxiv_tool]
)

# Agente 2: Analista
analyst = Agent(
    role='Analista de Datos',
    goal='Analizar la información recopilada e identificar insights clave',
    backstory='Especialista en análisis cuantitativo y cualitativo',
    tools=[python_repl_tool]
)

# Agente 3: Escritor
writer = Agent(
    role='Escritor Técnico',
    goal='Crear un reporte claro y bien estructurado sobre {topic}',
    backstory='Escritor con expertise en comunicación científica',
    tools=[grammar_checker_tool]
)
```

Ejemplo: CrewAI - Definir Tareas

```
# Tarea 1: Investigar
research_task = Task(
    description='Investiga las últimas tendencias en {topic}. '
    'Busca papers recientes, casos de uso, y estadísticas.',
    agent=researcher,
    expected_output='Reporte detallado con al menos 10 fuentes confiables'
)

# Tarea 2: Analizar
analysis_task = Task(
    description='Analiza la información del researcher. '
    'Identifica patrones, tendencias y puntos clave.',
    agent=analyst,
    expected_output='Análisis estructurado con insights principales',
    context=[research_task] # Depende de research_task
)

# Tarea 3: Escribir
writing_task = Task(
    description='Escribe un artículo de 1000 palabras sobre {topic} '
    'basado en la investigación y análisis.',
    agent=writer,
    expected_output='Artículo bien estructurado con introducción, desarrollo y conclusión',
    context=[research_task, analysis_task] # Depende de ambas anteriores
)

# Crear el crew y ejecutar
crew = Crew(
    agents=[researcher, analyst, writer],
    tasks=[research_task, analysis_task, writing_task],
    verbose=True
)

result = crew.kickoff(inputs={'topic': 'Quantum Computing en 2025'})
```

Tendencias 2024-2025 en Multi-Agentes

La industria está evolucionando rápidamente:

1. Microsoft Agent Framework (2025)

- Unifica Semantic Kernel y AutoGen
- Soporte para enterprise: escalabilidad, seguridad, observabilidad
- Workflow orchestration determinístico + agent orchestration creativo

2. Google Agent2Agent Protocol (A2A, 2025)

- Protocolo para comunicación entre agentes de diferentes plataformas
- Similar a MCP pero para coordinación entre agentes
- Permite que un agente de OpenAI colabore con uno de Google

3. OpenAI Agents SDK (Marzo 2025)

- Framework ligero para multi-agent workflows
- Enfoque en tracing y guardrails
- Integrado con GPT-4 y o1

No todo es color de rosa. Desafíos actuales:

1. Coordinación y sincronización

- ¿Cómo evitar que agentes trabajen en tareas duplicadas?
- ¿Qué pasa si dos agentes necesitan el mismo recurso?
- Deadlocks y race conditions son posibles

2. Costo exponencial

- Cada agente hace llamadas al LLM
- 4 agentes → 4x el costo (o más si iteran)
- Necesitas balance entre calidad y presupuesto

3. Debugging y observabilidad

- Difícil rastrear qué agente causó un error
- Logs distribuidos entre múltiples agentes
- Herramientas de tracing aún inmaduras

Best Practices para Multi-Agentes

1. Diseño de agentes

- Roles claros y no superpuestos. Cada agente debe tener expertise definida
- Evitar agentes "genéricos" que hacen de todo

2. Comunicación

- Usar un shared context store (vector DB)
- Mensajes estructurados con schemas claros
- Logs detallados de cada interacción

3. Control de flujo

- Límite máximo de iteraciones por agente
- Timeouts para cada tarea
- Circuit breakers si un agente falla repetidamente

4. Evaluación

- Medir success rate del sistema completo
- Trackear costo por tarea (tokens usados)

Comparación: Single vs Multi-Agente

Agente Único

Ventajas:

- Más simple de implementar
- Menor costo (1 agente)
- Debugging más fácil
- Menos latencia

Limitaciones:

- Tareas complejas = prompts enormes
- Sin especialización
- Sin validación cruzada
- Difícil escalar complejidad

Multi-Agente

Ventajas:

- Maneja tareas muy complejas
- Especialización por área
- Auto-validación y crítica
- Paralelización

Limitaciones:

- Más complejo de diseñar
- Mayor costo (N agentes)
- Debugging difícil
- Mayor latencia total

Transición a la Siguiente Sección

Hemos cubierto:

- Por qué necesitamos sistemas multi-agente
- Patrones de comunicación y coordinación
- Caso real: sistema de investigación de Anthropic
- Frameworks populares (AutoGen, CrewAI, LangGraph)
- Tendencias de la industria (2024-2025)
- Desafíos y best practices

Lo que viene

Ahora exploraremos las **fronteras de investigación en LLMs**: reasoning models (o1, DeepSeek-R1), test-time compute, nuevas arquitecturas, y hacia dónde se dirige el campo.

Outline : Fronteras de Investigación en LLMs

El Viaje hasta Aquí: De
Transformers a Agentes

¿Qué es un Agente?

Function Calling y Tool Use

Model Context Protocol (MCP)

Sistemas Multi-Agente

Fronteras de Investigación en
LLMs

Test-Time Compute: Pensar Más = Mejor Resultado

Concepto

Invertir más cómputo durante **inferencia** (no entrenamiento) para mejorar la calidad de las respuestas.

Técnicas principales:

1. Chain-of-Thought extenso:

- Permitir razonamientos mucho más largos
- OpenAI o1: 10k-100k tokens de "pensamiento"

2. Self-consistency (votación):

- Generar múltiples soluciones
- Votar por la mejor

3. Tree search durante inference:

- Explorar múltiples caminos
- Evaluar y seleccionar mejores

Trade-off: Mayor latencia y costo, pero mejor precisión en tareas difíciles.

OpenAI o1: Large Reasoning Models

OpenAI o1 (lanzado Septiembre 2024) es un cambio de paradigma:

- **No es solo un LLM más grande**
- Entrenado con **Reinforcement Learning** para razonar
- Genera "cadenas de pensamiento" internas muy largas
- El modelo "piensa" antes de responder (hasta minutos)

Resultados impresionantes:

Benchmark	GPT-4o	o1-preview
AIME (matemáticas olímpicas)	13.4%	74.4%
Codeforces (programación)	11th percentile	89th percentile
PhD-level physics	-	Comparable a PhDs

Implicación

El futuro puede estar en **modelos que piensan más**, no solo en modelos más grandes.

Resultados OpenAI o1

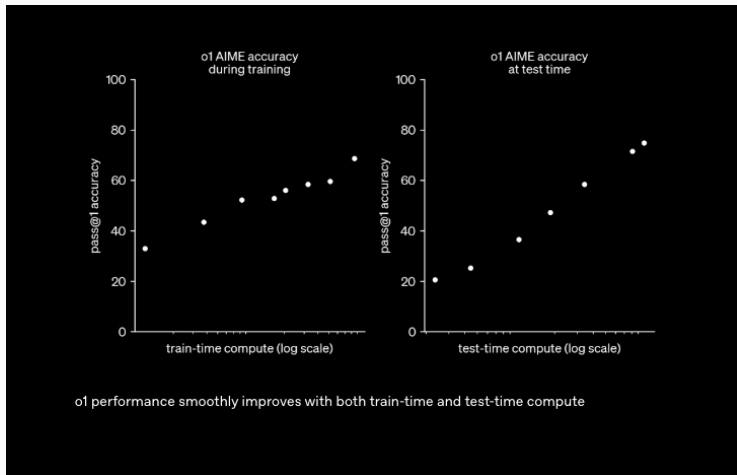


Figure 5: Resultados Reportados por OpenAI para O1

DeepSeek-R1: Open Reasoning Model

DeepSeek-R1 (Enero 2025): Primera alternativa open source competitiva a o1.

- Entrenado con **RL puro** (sin supervised fine-tuning)
- Capacidades de razonamiento emergentes
- Distilled versions: R1-Distill-Qwen (7B, 14B, 32B)

Performance:

- AIME: 79.8% (supera a o1-preview)
- MATH-500: 97.3%
- Comparable a o1 en muchos benchmarks

Insight clave

RL puede enseñar a modelos a razonar sin necesidad de datos de razonamiento explícitos. El modelo **descubre** estrategias de razonamiento por sí solo.

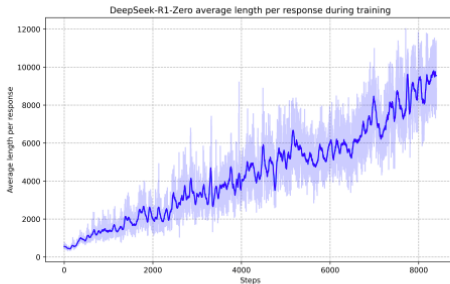


Figure 3 | The average response length of DeepSeek-R1-Zero on the training set during the RL process. DeepSeek-R1-Zero naturally learns to solve reasoning tasks with more thinking time.

Multi-Token Prediction: El Futuro del Training

Paper: Gloeckle et al. (Meta, 2024) - "Better & Faster Large Language Models via Multi-Token Prediction"

Problema con next-token prediction

Entrenar para predecir solo el siguiente token puede ser **subóptimo** para aprender razonamiento de largo plazo.

Solución: Multi-Token Prediction (MTP)

- Predecir los próximos n tokens simultáneamente (ej: $n = 4$)
- Cada posición tiene su propia cabeza de predicción
- Fuerza al modelo a planificar a futuro

Beneficios observados:

1. **Mejor en code:** +17% en HumanEval
2. **Inferencia más rápida:** Genera múltiples tokens por forward pass
3. **Mejor razonamiento largo:** Planificación mejorada

Multi-token Prediction

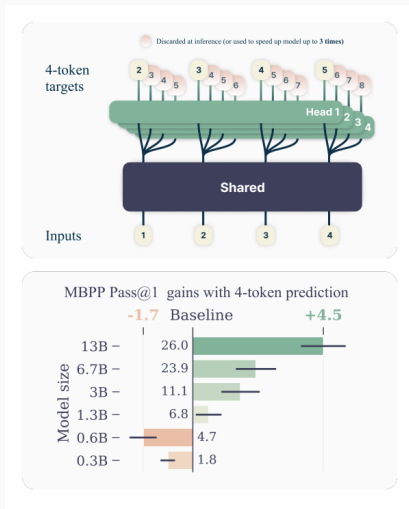


Figure 6: Figure 1: Overview of multi-token prediction. (Top) During training, the model predicts 4 future tokens at once, by means of a shared trunk and 4 dedicated output heads. During inference, we employ only the next-token

Multi-Token Prediction: Arquitectura

Arquitectura tradicional:

Input tokens \rightarrow Transformer \rightarrow 1 prediction head \rightarrow next token

Arquitectura con MTP:

Input tokens \rightarrow Shared Transformer \rightarrow

- Head 1 \rightarrow token $t + 1$
- Head 2 \rightarrow token $t + 2$
- Head 3 \rightarrow token $t + 3$
- Head 4 \rightarrow token $t + 4$

Durante training:

- Loss combinado de todas las cabezas. El modelo aprende dependencias a futuro

Durante inference:

- Usar todas las predicciones (speculative decoding)
- Verificar y aceptar tokens correctos. 2-3x speedup en generación

Computer Use: Agentes que Controlan tu PC

Anthropic Computer Use (Octubre 2024): Claude puede controlar computadoras. **demo**

- Toma screenshots de la pantalla
- Decide dónde hacer click, escribir, etc.
- Ejecuta acciones via API
- Loop: observa → piensa → actúa

Ejemplo de tarea:

User: "Busca información sobre el PIB de Chile y créame una presentación con los datos"

Agent:

1. Abre browser → screenshot
2. Va a Google → screenshot
3. Busca "PIB Chile 2024" → screenshot
4. Lee resultados → extrae datos
5. Abre PowerPoint → screenshot
6. Crea slides con datos → screenshot
7. Guarda presentación

Limitaciones actuales: Lento, puede cometer errores, requiere supervisión.

Long-term Memory y Aprendizaje Continuo

Desafío actual: Los agentes actuales no "aprenden" de interacciones pasadas.

- Cada sesión es independiente
- No mejoran con uso
- Olvidan contexto entre conversaciones

Direcciones de investigación:

1. Episodic Memory:

- Almacenar experiencias pasadas
- Retrieval semántico cuando sea relevante
- Voyager (Minecraft agent) usa esto

2. In-context RL:

- Aprender de feedback en contexto
- Sin actualizar pesos del modelo

3. Continual Learning:

- Fine-tuning incremental del agente
- Retención de conocimiento previo
- Problema: catastrophic forgetting

Agentic Workflows en Producción

Desafío: Pasar de prototipos a sistemas confiables en producción.

Consideraciones clave:

1. Confiabilidad:

- Manejo robusto de errores
- Fallbacks cuando API falla
- Validación de outputs

2. Observabilidad:

- Logging detallado de cada step
- Tracing de chains (LangSmith, Weights & Biases)
- Métricas de performance

3. Costos:

- Caché de resultados
- Uso de modelos más baratos cuando sea posible
- Límites de tokens por request

4. Seguridad:

- Validación de tools disponibles
- Sandboxing de code execution
- Rate limiting

Limitaciones y Riesgos de Agentes Autónomos

Limitaciones técnicas:

- **Alucinaciones:** Pueden generar información falsa
- **Falta de common sense:** Errores básicos de lógica
- **Context limits:** Olvidan información en conversaciones largas
- **Cost:** Muchas llamadas a API pueden ser caras

Riesgos de seguridad:

- **Prompt injection:** Manipulación vía inputs maliciosos
- **Tool misuse:** Uso inadecuado de herramientas peligrosas
- **Data leakage:** Exposición de información sensible
- **Runaway agents:** Loops infinitos o comportamiento inesperado

Principio de seguridad

Nunca dar a un agente más permisos de los necesarios. Siempre validar outputs críticos antes de ejecutar acciones irreversibles.

Preguntas fundamentales:

1. **¿Quién es responsable de las acciones del agente?**
 - El desarrollador? El usuario? El modelo?
2. **¿Cómo asegurar que agentes sigan valores humanos?**
 - Constitutional AI (Anthropic)
 - RLHF (Reinforcement Learning from Human Feedback)
3. **¿Transparencia vs autonomía?**
 - ¿Deben los agentes explicar sus decisiones?
 - ¿O solo importa el resultado?

Trabajo activo

Investigación intensa en **alignment**, **interpretability**, y **safe agents**.
Área crítica a medida que agentes se vuelven más autónomos.

Conceptos Clave para Recordar

1. **Agentes = LLM + Memory + Tools + Planning**
 - No es solo un chat, es un sistema completo
2. **Function calling es fundamental**
 - Permite interacción con mundo externo
3. **ReAct ¿ Simple CoT para tareas complejas**
 - Intercalar razonamiento con acciones
4. **Test-time compute mejora razonamiento**
 - Pensar más = mejores resultados
5. **Multi-agente es el futuro**
 - Especialización y colaboración
6. **MCP: el USB de los LLMs**
 - Protocolo estándar para herramientas
7. **Evaluación de agentes es difícil**
 - Success rate, cost, reliability

Tips Prácticos para Construir Agentes

Diseño de Tools:

- Descripciones claras y específicas
- Validación robusta de inputs
- Manejo de errores con mensajes informativos
- Logging detallado de todas las llamadas

Prompts para Agentes:

- Instrucciones claras sobre cuándo usar cada tool
- Ejemplos de uso correcto (few-shot)
- Límites explícitos (máx iteraciones, presupuesto)
- Formato de output bien definido

Debugging:

- Activar `verbose=True` para ver trace completo
- Loggear cada step del agente
- Probar tools individualmente primero y limitar iteraciones máximas para evitar loops

Agentes, Tools y Fronteras de LLMs

¿Dudas sobre algún concepto?

Próxima clase: Laboratorio práctico

Questions?



H. Face.

Deep dive into text generation inference with llms, 2025.

Accedido: 25 de noviembre de 2025.



H. Face.

What are agents? — conceptual guides for smolagents, 2025.

Accedido: 25 de noviembre de 2025.