IACADEMY

AGENTIC ENGINEER & SDD

MÓDULO 04

# Tools y Function Calling

Avanzado

iacedemy.com — 2026

MÓDULO 04

# Tools y Function Calling

**Nivel:** Avanzado

**Autor:** Alicia Sanz

**Publicación:** Julio 2026

**Plataforma:** iacedemy.com

> “Las tools son las manos del agente. Manos mal disenadas = agente torpe.”

---

# 4.1 Cómo Funciona Function Calling

## El Mecanismo

```
1. Tu defines tools como JSON Schema
2. Envias el schema al LLM junto con el mensaje del usuario
3. El LLM decide si necesita una tool y cual
4. El LLM genera un JSON con el nombre de la tool y los argumentos
5. TU ejecutas la tool (el LLM NO la ejecuta)
6. Devuelves el resultado al LLM
7. El LLM decide si necesita otra tool o si ya puede responder
```

```python
# Flujo completo de function calling

# Paso 1: Definir tool schema
tools = [{
    "type": "function",
    "function": {
        "name": "search_knowledge_base",
        "description": "Buscar articulos en la base de conocimiento por
query semantica. Usar cuando el usuario pregunta algo que podria estar
documentado.",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "Query de búsqueda en lenguaje
natural"
                },
                "top_k": {
                    "type": "integer",
                    "description": "Numero de resultados a devolver",
                    "default": 5
                }
            },
            "required": ["query"]
        }
    }
}]

# Paso 2: Enviar al LLM
response = llm.complete(
    messages=[
        {"role": "system", "content": "Eres un asistente de soporte..."},
        {"role": "user", "content": "Cómo reseteo mi password?"}
    ],
    tools=tools
)

# Paso 3: El LLM responde con tool_call
# response.tool_calls = [
#   ToolCall(
#     id="call_abc123",
#     name="search_knowledge_base",
#     arguments={"query": "resetear password", "top_k": 3}
#   )
# ]

# Paso 4: TU ejecutas la tool
result = search_knowledge_base(
```

```
    query="resetear password",
    top_k=3
)

# Paso 5: Devolver resultado al LLM
messages.append({"role": "assistant", "content": None, "tool_calls":
response.tool_calls})
messages.append({"role": "tool", "tool_call_id": "call_abc123",
"content": str(result)})

# Paso 6: El LLM genera respuesta final usando el resultado
final_response = llm.complete(messages=messages, tools=tools)
```

## JSON Schema para Tools

```python
# Anatomia de un tool schema bien escrito

tool_schema = {
    "type": "function",
    "function": {
        "name": "create_support_ticket",

        # DESCRIPCION: critica. El LLM decide cuando usar la tool
basandose en esto.
        # Incluir: que hace, cuando usarla, que NO hacer con ella.
        "description": (
            "Crear un ticket de soporte para el usuario actual. "
            "Usar cuando el problema del usuario no se puede resolver "
            "con la base de conocimiento y necesita atencion humana. "

"NO usar para preguntas frecuentes que tienen respuesta en la KB."
        ),

        "parameters": {
            "type": "object",
            "properties": {
                "title": {
                    "type": "string",
                    "description": "Titulo corto del problema (max 100
chars)",
                    "maxLength": 100
                },
                "description": {
                    "type": "string",
                    "description": "Descripcion detallada incluyendo
pasos para reproducir, error exacto, y que ya se intento"
                },
                "priority": {
                    "type": "string",
                    "enum": ["low", "medium", "high", "urgent"],
                    "description": "low: pregunta general. medium:
problema que afecta al trabajo. high: bloqueado. urgent: caida de
servicio."
                },
                "category": {
                    "type": "string",
                    "enum": ["billing", "technical", "account", "feature_
request"],
                    "description": "Categoria del ticket para routing
automatico"
                }
            },
            "required": ["title", "description", "priority", "category"]
        }
```

```
    }
}
```

---

## 4.2 Escribir Buenos Schemas

### Regla 1: Descripcion Orientada al Uso

```
# MAL: descripcion tecnica
"description": "Ejecuta SELECT query contra la tabla knowledge_base en
PostgreSQL"

# BIEN: descripcion orientada a cuando usarla
"description": "Buscar informacion en la base de conocimiento. Usar
cuando el usuario hace una pregunta que podria estar documentada.
Devuelve los articulos más relevantes con su contenido y fuente."

# MEJOR: con ejemplos de cuando SI y cuando NO
"description": (
    "Buscar informacion en la base de conocimiento. "
    "USAR cuando: pregunta sobre producto, error conocido, como hacer
algo. "

"NO USAR cuando: el usuario pide hablar con humano, pregunta sobre su
cuenta especifica, "
    "o la pregunta no tiene relacion con nuestro producto."
)
```

## Regla 2: Enums sobre Strings Libres

```
# MAL: string libre, el LLM puede inventar cualquier valor
"priority": {
    "type": "string",
    "description": "La prioridad del ticket"
}
# El LLM podria generar: "muy alta", "ASAP", "5", "critical", etc.

# BIEN: enum con valores definidos y descritos
"priority": {
    "type": "string",
    "enum": ["low", "medium", "high", "urgent"],
    "description": "low=pregunta general, medium=afecta trabajo,
high=bloqueado, urgent=caida servicio"
}
```

## Regla 3: Defaults Explicitos

```
# MAL: sin default, el LLM no sabe que pasar si el usuario no especifica
"top_k": {
    "type": "integer",
    "description": "Numero de resultados"
}

# BIEN: default explicito
"top_k": {
    "type": "integer",
    "description": "Numero de resultados a devolver. Default 5 es
suficiente para la mayoria de queries.",
    "default": 5,
    "minimum": 1,
    "maximum": 20
}
```

## Regla 4: Validacion en el Schema

```
"parameters": {
    "type": "object",
    "properties": {
        "email": {
            "type": "string",
            "format": "email",
            "description": "Email del usuario"
        },
        "amount": {
            "type": "number",
            "minimum": 0.01,
            "maximum": 10000,
            "description": "Monto en EUR"
        },
        "date": {
            "type": "string",
            "format": "date",
            "description": "Fecha en formato YYYY-MM-DD"
        },
        "tags": {
            "type": "array",
            "items": {"type": "string"},
            "maxItems": 5,
            "description": "Tags para categorizar (max 5)"
        }
    }
}
```

## 4.3 Patron: Tool Registry

```python
"""
Registro centralizado de tools con metadata, validacion y middleware.
"""

from dataclasses import dataclass, field
from typing import Callable, Any
from enum import Enum
import json
import time

class RiskLevel(Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
    CRITICAL = "critical"

@dataclass
class ToolDefinition:
    name: str
    description: str
    function: Callable
    parameters_schema: dict
    risk_level: RiskLevel = RiskLevel.LOW
    timeout_seconds: int = 10
    requires_hitl: bool = False
    idempotent: bool = True
    cost_per_call: float = 0.0
    rate_limit: int | None = None  # calls per minute

    _call_count: int = field(default=0, init=False)
    _last_reset: float = field(default_factory=time.time, init=False)

class ToolRegistry:
    def __init__(self):
        self._tools: dict[str, ToolDefinition] = {}
        self._middleware: list[Callable] = []

    def register(self, tool: ToolDefinition):
        self._tools[tool.name] = tool

    def add_middleware(self, fn: Callable):
        """Middleware se ejecuta antes de cada tool call."""
        self._middleware.append(fn)

    async def execute(self, name: str, arguments: dict) -> dict:
        tool = self._tools.get(name)
        if not tool:
            return {"error": f"Tool '{name}' no registrada"}
```

```
        # Ejecutar middleware (logging, validacion, rate limiting)
        for middleware in self._middleware:
            result = await middleware(tool, arguments)
            if result is not None:  # Middleware puede bloquear
                return result

        # Rate limiting
        if tool.rate_limit:
            now = time.time()
            if now - tool._last_reset > 60:
                tool._call_count = 0
                tool._last_reset = now
            tool._call_count += 1
            if tool._call_count > tool.rate_limit:
                return {"error": f"Rate limit: max {tool.rate_limit}/
min"}

        # Ejecutar con timeout
        try:
            import asyncio
            result = await asyncio.wait_for(
                tool.function(**arguments),
                timeout=tool.timeout_seconds
            )
            return {"success": True, "data": result}
        except asyncio.TimeoutError:
            return {"error": f"Timeout tras {tool.timeout_seconds}s"}
        except Exception as e:
            return {"error": str(e)}

    def get_schemas(self) -> list[dict]:
        """Generar schemas para enviar al LLM."""
        return [{
            "type": "function",
            "function": {
                "name": tool.name,
                "description": tool.description,
                "parameters": tool.parameters_schema,
            }
        } for tool in self._tools.values()]


# Uso
registry = ToolRegistry()

# Middleware de logging
async def log_middleware(tool: ToolDefinition, args: dict):
    print(f"[TOOL] {tool.name}({json.dumps(args)})
risk={tool.risk_level.value}")
    return None  # None = continuar ejecucion
```

```python
# Middleware de HITL
async def hitl_middleware(tool: ToolDefinition, args: dict):
    if tool.requires_hitl:
        approved = await request_human_approval(tool.name, args)
        if not approved:
            return {"error": "Accion rechazada por el supervisor"}
    return None

registry.add_middleware(log_middleware)
registry.add_middleware(hitl_middleware)
```

---

# 4.4 MCP: Construir un Server Completo

## Server con Multiples Tools

```python
# mcp_knowledge_server.py
"""
MCP Server para gestion de conocimiento.
Expone tools de búsqueda, creacion y actualizacion de articulos.
"""

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

server = Server("knowledge-manager")
db_pool = None

@server.list_tools()
async def list_tools() -> list[Tool]:
    return [
        Tool(
            name="search_articles",
            description=(
                "Buscar articulos en la base de conocimiento por query
semantica. "
                "Devuelve titulo, extracto y URL de los más relevantes."
            ),
            inputSchema={
                "type": "object",
                "properties": {
                    "query": {"type": "string", "description": "Busqueda
en lenguaje natural"},
                    "category": {
                        "type": "string",
                        "enum": ["all", "faq", "troubleshooting",
"howto", "release_notes"],
                        "description": "Filtrar por categoria",
                        "default": "all"
                    },
                    "limit": {"type": "integer", "default": 5, "minimum":
 1, "maximum": 20}
                },
                "required": ["query"]
            }
        ),
        Tool(
            name="get_article",

description="Obtener el contenido completo de un articulo por ID.",
            inputSchema={
                "type": "object",
                "properties": {
                    "article_id": {"type": "string", "description": "ID
```

```
del articulo"}
                },
                "required": ["article_id"]
            }
        ),
        Tool(
            name="list_categories",
            description="Listar todas las categorias disponibles con el
numero de articulos en cada una.",
            inputSchema={
                "type": "object",
                "properties": {}
            }
        ),
    ]

@server.call_tool()
async def call_tool(name: str, arguments: dict) -> list[TextContent]:
    if name == "search_articles":
        return await _search_articles(**arguments)
    elif name == "get_article":
        return await _get_article(**arguments)
    elif name == "list_categories":
        return await _list_categories()
    else:
        return [TextContent(type="text", text=f"Tool desconocida:
{name}")]

async def _search_articles(query: str, category: str = "all", limit: int
= 5):
    sql = """
        SELECT id, title,
               ts_rank(search_vector, websearch_to_tsquery('spanish',
$1)) as rank,
               LEFT(content, 200) as excerpt
        FROM articles
        WHERE ($2 = 'all' OR category = $2)
          AND search_vector @@ websearch_to_tsquery('spanish', $1)
        ORDER BY rank DESC
        LIMIT $3
    """
    rows = await db_pool.fetch(sql, query, category, limit)

    if not rows:
        return [TextContent(type="text", text=f"No se encontraron
articulos para: {query}")]

    results = []
    for row in rows:
        results.append(f"[{row['id']}] {row['title']}\n{row['excerpt']}..
```

```
.")

    return [TextContent(type="text", text="\n\n---\n\n".join(results))]

async def _get_article(article_id: str):
    row = await db_pool.fetchrow(
        "SELECT title, content, category, updated_at FROM articles WHERE
id = $1",
        article_id
    )
    if not row:
        return [TextContent(type="text", text=f"Articulo {article_id} no
encontrado")]

    return [TextContent(
        type="text",
        text=f"# {row['title']}\nCategoria: {row['category']}\nActualizad
o: {row['updated_at']}\n\n{row['content']}"
    )]

async def _list_categories():
    rows = await db_pool.fetch(
        "SELECT category, COUNT(*) as count FROM articles GROUP BY
category ORDER BY count DESC"
    )
    lines = [f"- {row['category']}: {row['count']} articulos" for row in
rows]
    return [TextContent(type="text", text="Categorias:\n" + "\n".join(lin
es))]

# Startup
async def main():
    global db_pool
    db_pool = await asyncpg.create_pool("postgresql://...")

    from mcp.server.stdio import stdio_server
    async with stdio_server() as (read, write):
        await server.run(read, write)

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

## Configurar el MCP Client

```
// .mcp.json (o claude_desktop_config.json)
{
  "mcpServers": {
    "knowledge-manager": {
      "command": "python",
      "args": ["mcp_knowledge_server.py"],
      "env": {
        "DATABASE_URL": "postgresql://user:pass@localhost/kb"
      }
    }
  }
}
```

---

# 4.5 Error Handling en Tools

## Patron: Resultado Estructurado

```python
from dataclasses import dataclass
from typing import Any

@dataclass
class ToolResult:
    """
    Resultado estandar de tool. El agente siempre sabe:
    - Si funciono (success)
    - Que datos tiene (data)
    - Que salio mal (error)
    - Que deberia hacer a continuacion (suggestion)
    """
    success: bool
    data: Any = None
    error: str | None = None
    suggestion: str | None = None

    def to_llm_string(self) -> str:
        """Formato optimizado para el LLM."""
        if self.success:
            return f"OK: {json.dumps(self.data, ensure_ascii=False)}"

        msg = f"ERROR: {self.error}"
        if self.suggestion:
            msg += f"\nSugerencia: {self.suggestion}"
        return msg
```

## Errores Comunes y Cómo Manejarlos

```
async def robust_tool_execution(tool_name: str, params: dict) ->
ToolResult:
    """Wrapper con manejo de errores estandar."""

    try:
        result = await execute_tool(tool_name, params)
        return ToolResult(success=True, data=result)

    except TimeoutError:
        return ToolResult(
            success=False,
            error=f"{tool_name} tardo más de {timeout}s",
            suggestion="Intentar con parámetros más restrictivos o
reintentar en 30s"
        )

    except PermissionError:
        return ToolResult(
            success=False,
            error=f"Sin permisos para ejecutar {tool_name}",
            suggestion="Escalar a un usuario con permisos adecuados"
        )

    except ConnectionError:
        return ToolResult(
            success=False,
            error=f"No se pudo conectar al servicio de {tool_name}",
            suggestion="El servicio puede estar temporalmente no
disponible. Informar al usuario y reintentar."
        )

    except ValidationError as e:
        return ToolResult(
            success=False,
            error=f"Parametros invalidos: {e}",
            suggestion="Revisar los parámetros y corregir el formato"
        )

    except Exception as e:
        return ToolResult(
            success=False,
            error=f"Error inesperado: {type(e).__name__}: {str(e)}",
            suggestion="Error no esperado. Escalar a soporte tecnico."
        )
```

# 4.6 Parallel Tool Calls

## Cuándo Usar Ejecucion Paralela

```
PARALELO (independientes):
  - search_kb("error login") + get_user_history(user_id)
  - check_status("api") + check_status("db") + check_status("cache")
  Los resultados no dependen entre si.

SECUENCIAL (dependientes):
  - search_kb("error") → resultado → create_ticket(basado_en_resultado)
  - get_user(id) → user → update_user(user.email, new_data)
  El segundo depende del primero.
```

## Implementacion

```
import asyncio

async def execute_parallel_tools(
    tool_calls: list[dict],
    registry: ToolRegistry
) -> list[dict]:
    """Ejecutar multiples tool calls en paralelo."""

    tasks = [
        registry.execute(tc["name"], tc["arguments"])
        for tc in tool_calls
    ]

    results = await asyncio.gather(*tasks, return_exceptions=True)

    return [
        {
            "tool_call_id": tc["id"],
            "result": r if not isinstance(r, Exception) else {"error": st
r(r)}
        }
        for tc, r in zip(tool_calls, results)
    ]
```

## Soporte por Provider

```
OpenAI:    Soporta parallel tool calls nativamente
Anthropic: Soporta parallel tool calls nativamente
Qwen:      Depende del servidor (vLLM soporta)
Llama:     Depende del fine-tune y formato

Si el modelo no soporta parallel nativo,
tu puedes detectar tools independientes y paralelizarlas tu.
```

---

# 4.7 Tool Output: Formato para el LLM

## Regla: Outputs Concisos, No Raw

```
# MAL: dump de JSON crudo con datos innecesarios
{
    "id": "art_123",
    "title": "Cómo resetear password",
    "content": "...",
    "created_at": "2024-01-15T10:30:00Z",
    "updated_at": "2024-06-20T14:22:00Z",
    "author_id": "usr_456",
    "author_name": "Ana Garcia",
    "version": 7,
    "word_count": 342,
    "views": 1523,
    "internal_tags": ["password", "reset", "security"],
    "internal_notes": "Revisar tras migracion auth v3",
    "status": "published",
    "seo_slug": "como-resetear-password",
    # ... 20 campos más que el LLM no necesita
}

# BIEN: solo lo relevante para responder al usuario
"""
Articulo encontrado: "Cómo resetear password"
Fuente: https://help.example.com/password-reset

Pasos:
1. Ir a Configuracion > Seguridad
2. Clic en "Cambiar contrasena"
3. Introducir contrasena actual y nueva
4. Confirmar con email de verificacion

Nota: Si no recuerdas la contrasena actual, usar "Olvide mi contrasena"
en login.
"""
```

### Tamano del Output

```
REGLA: Tool result < 1000 tokens para contexto.
       Si el resultado es más grande, resumir o paginar.

EXCEPCIONES:
  - Si el agente necesita el dato completo para razonar (ej: codigo
fuente)
  - Si el resultado se va a guardar en scratchpad, no en contexto

IMPLEMENTACION:
  def format_for_llm(data, max_tokens=800):
      formatted = json.dumps(data, ensure_ascii=False)
      if count_tokens(formatted) > max_tokens:
          return summarize(formatted, max_tokens)
      return formatted
```

---

## 4.8 Ejercicios Practicos

### Ejercicio 1: Escribir Tool Schemas

Escribir JSON Schema para estas tools: 1. `search_products`: buscar productos por nombre, categoria, y rango de precio 2. `get_order_status`: consultar estado de un pedido por numero de orden 3. `submit_refund_request`: solicitar reembolso (requiere motivo y monto)

Incluir descripciones orientadas al LLM, enums, defaults, y validacion.

### Ejercicio 2: MCP Server

Crear MCP server para una agenda de contactos: - `list_contacts`: listar contactos con filtro por nombre - `get_contact`: obtener detalle de un contacto - `add_contact`: anadir nuevo contacto (nombre, email, telefono, notas) - `search_contacts`: búsqueda semantica en notas de contactos

## Ejercicio 3: Error Handling

Dado un agente que usa 3 tools (search, create, notify): - Implementar ToolResult estandar - Manejar: timeout, permission denied, not found, rate limit - Cada error debe tener suggestion accionable para el agente

## Ejercicio 4: Parallel vs Sequential

Dado este flujo, identificar que se puede paralelizar: 1. Buscar en KB la pregunta del usuario 2. Obtener historial de tickets del usuario 3. Verificar si hay incidencias activas en el sistema 4. Crear ticket si no hay respuesta en KB 5. Enviar notificacion al equipo de soporte

---

*Modulo 4 v1.0 . Curso Agentic Engineer & SDD 2026*