FastMCP – Guión

Introducción

FastMCP es un framework para crear servidores MCP basado en el estándar FastAPI.

El protocolo MCP (Model Context Protocol) permite que una IA utilice funciones o herramientas externas siguiendo un formato estandarizado.

FastMCP usa FastAPI para crear las funciones externas y MCP para que la IA pueda llamarlas adecuadamente.

Desarrollo

El protocolo MCP estandariza la comunicación de una IA con herramientas externas. Define un contrato claro entre el modelo y las funciones disponibles en un servidor MCP. Permite que una IA use software real como si fuesen módulos plug-and-play, sin tener que entender su implementación.

FastMCP permite convertir funciones de Python en herramientas que un modelo de IA puede llamar. No es necesario escribir rutas ni crear endpoints manuales.

```
@mcp.tool()
def add(a: int, b: int) -> int:
    return a + b
```

FastMCP se basa en el uso de decoradores, que son anotaciones incluidas en la parte superior de las funciones y clases, de forma similar a SpringBoot. Con estos decoradores, FastMCP permite identificar qué elementos son accesibles para los modelos conectados.

Hay tres componentes principales: herramientas (tools), recursos (resources) e instrucciones (prompts).

Herramientas

Una herramienta es una función que un modelo puede ejecutar de forma estandarizada. FastMCP la convierte en una operación remita. la IA envía los parámetros, el servidor la ejecuta y devuelve el resultado.

```
@mcp.tool()
def divide(a: int, b: int) -> float:
    if b == 0:
        return "No se puede dividir por cero"
    return a / b
```

Un modelo puede listar las herramientas disponibles por un servidor, y para cada una sabe el nombre de una función, qué recibe y qué devuelve.
Al invocar una función, el servidor valida los parámetros, ejecuta la función y devuelve el resultado al modelo. Esto permite que el modelo pueda realizar acciones como consultar bases de datos, hacer llamadas a APIs o acceder a archivos.

Podemos añadir metadatos a una herramienta:

```
@mcp.tool(
    name="find_products",           # Custom tool name for the LLM
    description="Search the product catalog", # Custom description
```

```
    tags={"catalog", "search"},      # Optional tags for organization/filtering
    meta={"version": "1.2", "author": "product-team"}  # Custom metadata
)
def search_products_implementation(query: str, category: str | None = None) ->
list[dict]:
    """Internal function description (ignored if description is provided)."""
    print(f"Searching for '{query}' in category '{category}'")
    return [{"id": 2, "name": "Another Product"}]
```

Recursos

Los recursos son datos estructurados y accesibles que un modelo puede consultar sin necesidad de ejecutar una función.

```
@mcp.resource("data://config")
def get_config() -> dict:
    """Provides application configuration as JSON."""
    return {
        "theme": "dark",
        "version": "1.2.0",
        "features": ["tools", "resources"],
    }
```

Los recursos suelen representar configuraciones, documentos o estados del sistema. Cada recurso necesita un URI para poder ser llamado por un cliente.

Los recursos pueden ser documentos que se actualizan periódicamente. Para evitar el polling, el servidor puede mandar una notificación a los clientes conectados para avisarles de cuándo un recurso ha sido añadido o ha cambiado.

```
@mcp.resource("data://example")
def example_resource() -> str:
    return "Hello!"

# These operations trigger notifications:
mcp.add_resource(example_resource)  # Sends resources/list_changed notification
example_resource.disable()          # Sends resources/list_changed notification
example_resource.enable()           # Sends resources/list_changed notification
```

Al igual que las herramientas, podemos añadir anotaciones para identificar recursos que son de sólo lectura o si son idempotentes.

```
@mcp.resource(
    "data://config",
    annotations={
```

```
        "readOnlyHint": True,
        "idempotentHint": True
    }
)
def get_config() -> dict:
    """Get application configuration."""
    return {"version": "1.0", "debug": False}
```

Los recursos pueden depender de parámetros pasados por el usuario. Esto se consigue mediante Resource Templates:

```
# Template with multiple parameters and annotations
@mcp.resource(
    "repos://{owner}/{repo}/info",
    annotations={
        "readOnlyHint": True,
        "idempotentHint": True
    }
)
def get_repo_info(owner: str, repo: str) -> dict:
    """Retrieves information about a GitHub repository."""
    # In a real implementation, this would call the GitHub API
    return {
        "owner": owner,
        "name": repo,
        "full_name": f"{owner}/{repo}",
        "stars": 120,
        "forks": 48
    }
```

Instrucciones

Las instrucciones (prompts) son plantillas que el modelo puede solicitar para generar respuestas estructuradas. Son plantillas reutilizables que guían al modelo sin ejecutar código ni acceder a datos.

```
# Prompt returning a specific message type
@mcp.prompt
def generate_code_request(language: str, task_description: str) -> PromptMessage:
    """Generates a user message requesting code generation."""
    content = f"Write a {language} function that performs the following task:
{task_description}"
    return PromptMessage(role="user", content=TextContent(type="text", text=content))
```

Al igual que los otros componentes, las instrucciones pueden incluir anotaciones:

```python
@mcp.prompt(
    name="analyze_data_request",          # Custom prompt name
    description="Creates a request to analyze data with specific parameters",  # Custom
description
    tags={"analysis", "data"},            # Optional categorization tags
    meta={"version": "1.1", "author": "data-team"}  # Custom metadata
)
def data_analysis_prompt(
    data_uri: str = Field(description="The URI of the resource containing the data."),
    analysis_type: str = Field(default="summary", description="Type of analysis.")
) -> str:
    """This docstring is ignored when description is provided."""
    return f"Please perform a '{analysis_type}' analysis on the data found at
{data_uri}."
```

Características avanzadas

Además de los tres componentes principales, FastMCP incluye otras características.

Server Composition

Cambiar múltiples servidores MCP en una sola aplicación. Puede ser de forma estática import_server), por lo que se hace una copia de los componentes de cada servidores o dinámica (mount), donde el servidor principal delega peticiones en los servidores secundarios.

La composición de servidores permite mayor organización, modularidad y reusabilidad.

Context

Provee detalles sobre la sesión entre un cliente y un servidor MCP. Contiene datos de la solicitud, recursos disponibles y un registro de ejecución.

User Elicitation

Mecanismo que permite solicitar información extra al usuario, en caso de que no tenga suficientes datos para ejecutar un prompt o herramienta.

```python
@dataclass
class UserInfo:
    name: str
    age: int

@mcp.tool
async def collect_user_info(ctx: Context) -> str:
```

```
    """Collect user information through interactive prompts."""
    result = await ctx.elicit(
        message="Please provide your information",
        response_type=UserInfo
    )

    if result.action == "accept":
        user = result.data
        return f"Hello {user.name}, you are {user.age} years old"
    elif result.action == "decline":
        return "Information not provided"
    else:  # cancel
        return "Operation cancelled"
```

Logging

Permite enviar información al cliente. Sirve para depuración durante el desarrollo. Podemos enviar debug, info, warning o error.

```
@mcp.tool
async def analyze_data(data: list[float], ctx: Context) -> dict:
    """Analyze numerical data with comprehensive logging."""
    await ctx.debug("Starting analysis of numerical data")
    await ctx.info(f"Analyzing {len(data)} data points")

    try:
        if not data:
            await ctx.warning("Empty data list provided")
            return {"error": "Empty data list"}

        result = sum(data) / len(data)
        await ctx.info(f"Analysis complete, average: {result}")
        return {"average": result, "count": len(data)}

    except Exception as e:
        await ctx.error(f"Analysis failed: {str(e)}")
        raise
```

Middleware

Permite interceptar y modificar las peticiones y respuestas MCP. Se puede utilizar para autenticación, monitoreo, rate limiting, cacheado, manejo de errores.

Progress Reporting

Si una operación consume mucho tiempo, podemos utilizarla para indicar al usuario (mediante porcentajes o barras de carga) el estado de la operación.

```python
@mcp.tool
async def process_items(items: list[str], ctx: Context) -> dict:
    """Process a list of items with progress updates."""
    total = len(items)
    results = []

    for i, item in enumerate(items):
        # Report progress as we process each item
        await ctx.report_progress(progress=i, total=total)

        # Simulate processing time
        await asyncio.sleep(0.1)
        results.append(item.upper())

    # Report 100% completion
    await ctx.report_progress(progress=total, total=total)

    return {"processed": len(results), "results": results}
```

Proxy Servers

Servidores intermedios que permiten a un cliente interactuar con un servidor que reenvía las peticiones a otro servidor. Se utiliza para seguridad y aislamiento de sesiones.

Sampling

Permite generar múltiples posibles respuestas o ejecuciones de una herramienta a partir de un mismo input.

```python
@mcp.tool
async def generate_code_example(concept: str, ctx: Context) -> str:
    """Generate a Python code example for a given concept."""
    response = await ctx.sample(
        messages=f"Write a simple Python code example demonstrating '{concept}'.",
```

```
        system_prompt="You are an expert Python programmer. Provide concise, working
code examples without explanations.",
        temperature=0.7,
        max_tokens=300
    )

    code_example = response.text
    return f"```python\n{code_example}\n```"
```

Background Tasks

Operaciones de larga duración que se ejecutan asíncronamente..

Ejemplos

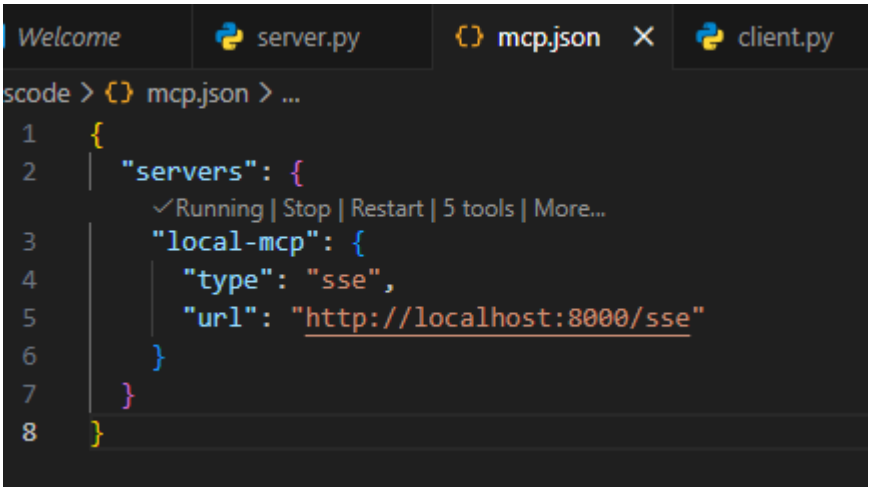
Ejemplo 1 – Servidor Básico

El primer ejemplo se basa en un servidor MCP básico, en el que exponemos herramientas, recursos e instrucciones.

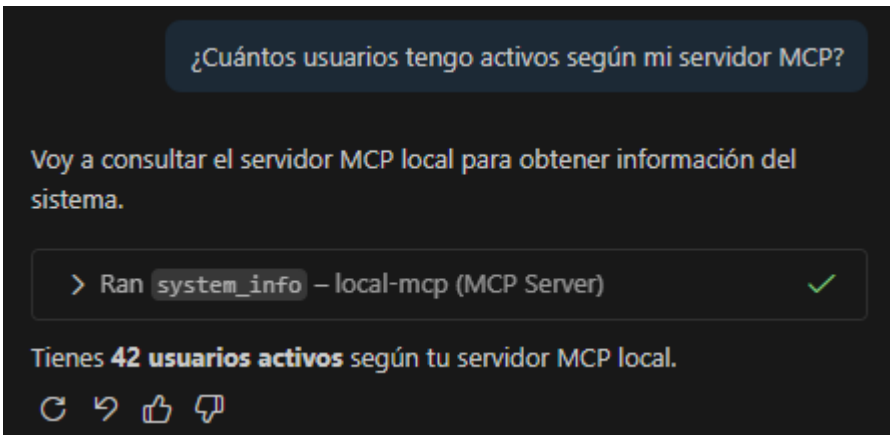
Podemos lanzar el servidor mediante python server.py, o bien crear y lanzar el contenedor docker siguiendo las instrucciones de docker.md.

Una vez lanzado el servidor, debemos conectarnos a él con un cliente. Un ejemplo de cliente se incluye en cliente.py, que llama al servidor y lista todo a lo que tiene acceso.

Para probar realmente el funcionamiento, se incluye otra carpeta desde la cuál vamos a acceder con Copilot. El acceso al servidor lo hacemos desde una carpeta diferente para que Copilot no analice los archivos internos del directorio, lo que le permitiría saber las herramientas disponibles sin acceder realmente al servidor.



Una vez Copilot está conectado al servidor, podemos abrir una sesión y hacer consultas relacionadas con las capacidades del servidor. Se incluyen ejemplos de consultas en el archivo consultas_mcp.md.



Para acceder correctamente a los recursos, podemos darle al icono de Add Context -> MCP Resources y seleccionar un recurso. El agente podrá utilizar el contenido de dicho recurso en la sesión.

Para utilizar correctamente los prompts, podemos ir a la carpeta .vscode -> mcp.json y, en el servidor al que queremos conectarnos, seleccionar en “prompts”. Podremos seleccionar uno de los prompts. VSCode nos pedirá los parámetros si fuese necesario.

Ejemplo 2 – Integración con OpenAPI

FastMCP se puede integrar con otros servicios mediante su especificación OpenAPI.

En el ejemplo 2 se incluyen dos servidores:

- El servidor MCP.
- Un servicio REST desarrollado en SpringBoot que se encarga de la gestión de un inventario de productos.

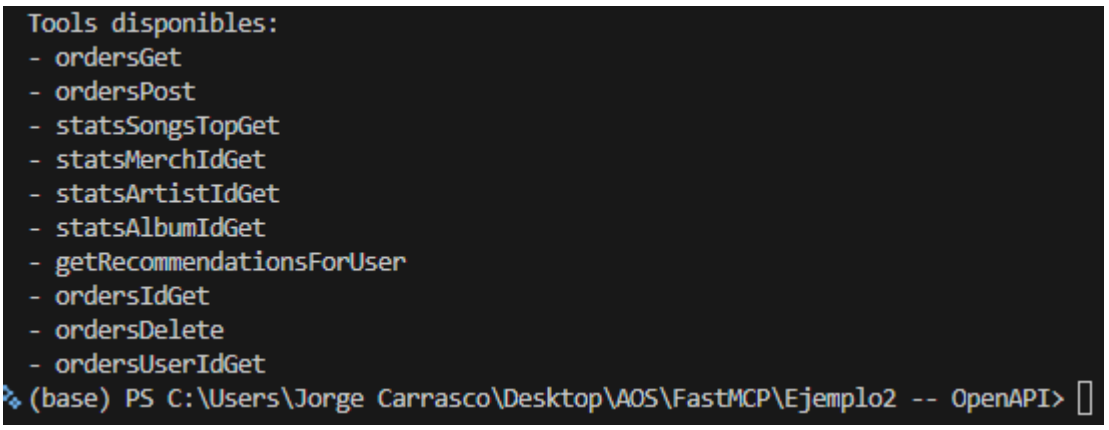
Podemos generar herramientas a partir de la especificación OpenAPI de un servicio REST. Para ello, necesitamos la especificación OpenAPI (accesible en /api-docs) y un cliente para hacer peticiones al servicio.

Para asegurar que se conecta adecuadamente en el contenedor, el servidor MCP intenta conectarse durante 1 minuto al servicio REST.

Una vez levantados los contenedores, podemos volver a acceder al servidor MCP desde la carpeta Acceso desde Copilot.

Entramos en .vscode/mcp.json y lanzamos el servidor local-mcp. Debería listar 8 herramientas disponibles.

Podemos hacer consultas al agente sobre los endpoints expuestos por el servicio Inventario. Por ejemplo, podemos pedirle listar los artículos disponibles, crear nuevos o modificar uno existente.



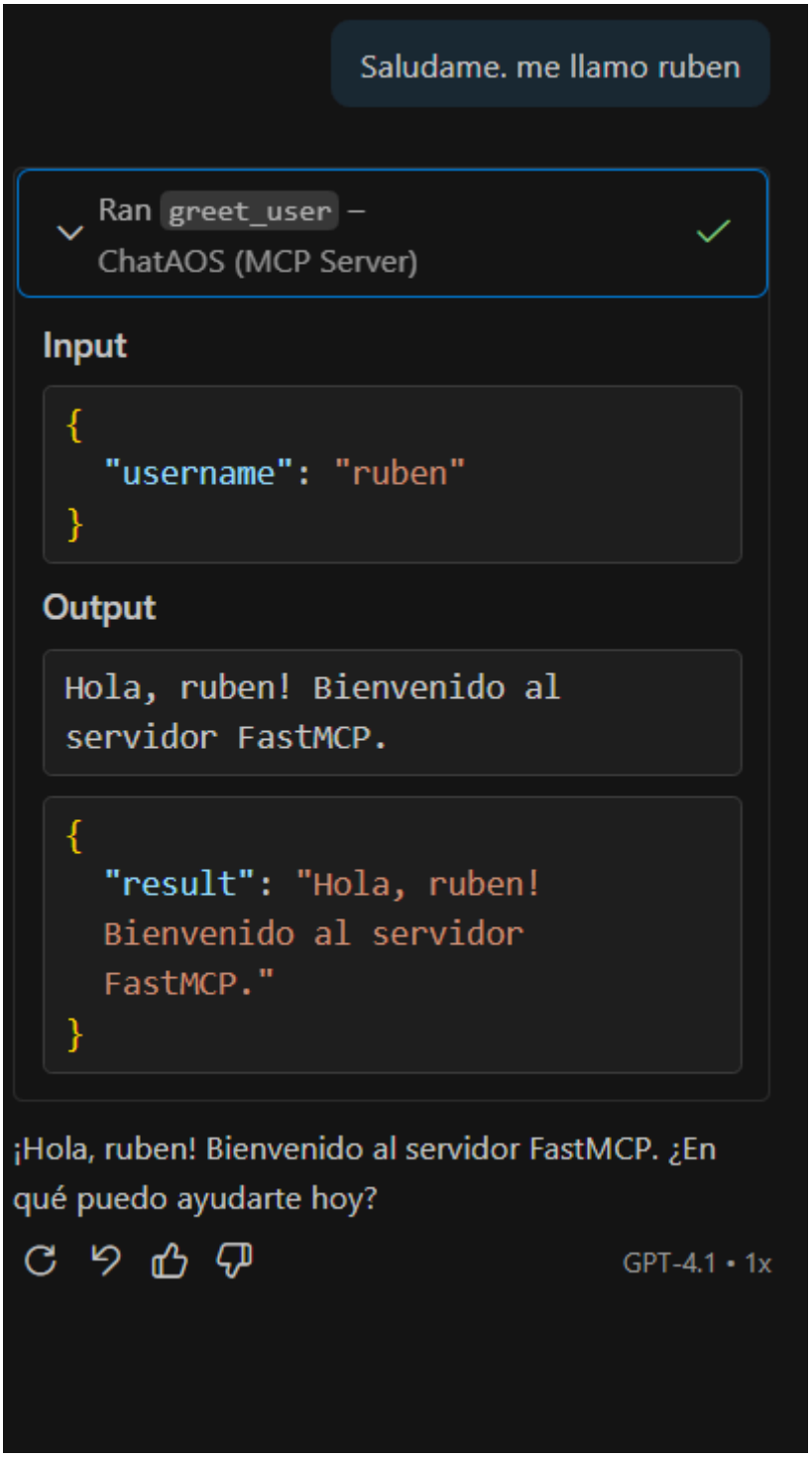
Ejemplo 3 – Integración con Gemini

En esta ocasión, se enfoca MCP como una herramienta para que LLM's realicen tareas específicas para las que no están diseñadas originalmente y que sean lo más precisas posibles. Tenemos 4 servidores.

Servidor 1

```python
from fastmcp import FastMCP

server = FastMCP("ChatAOS")

@server.tool()
def greet_user(username: str) -> str:
    return f"Hola, {username}! Bienvenido al servidor FastMCP."

if __name__ == "__main__":
    server.run()
```

En este ejemplo, se expone un tool de saludo, que simplemente devolverá un saludo básico cuando lo solicitemos con el formato que ahí le proporcionamos.

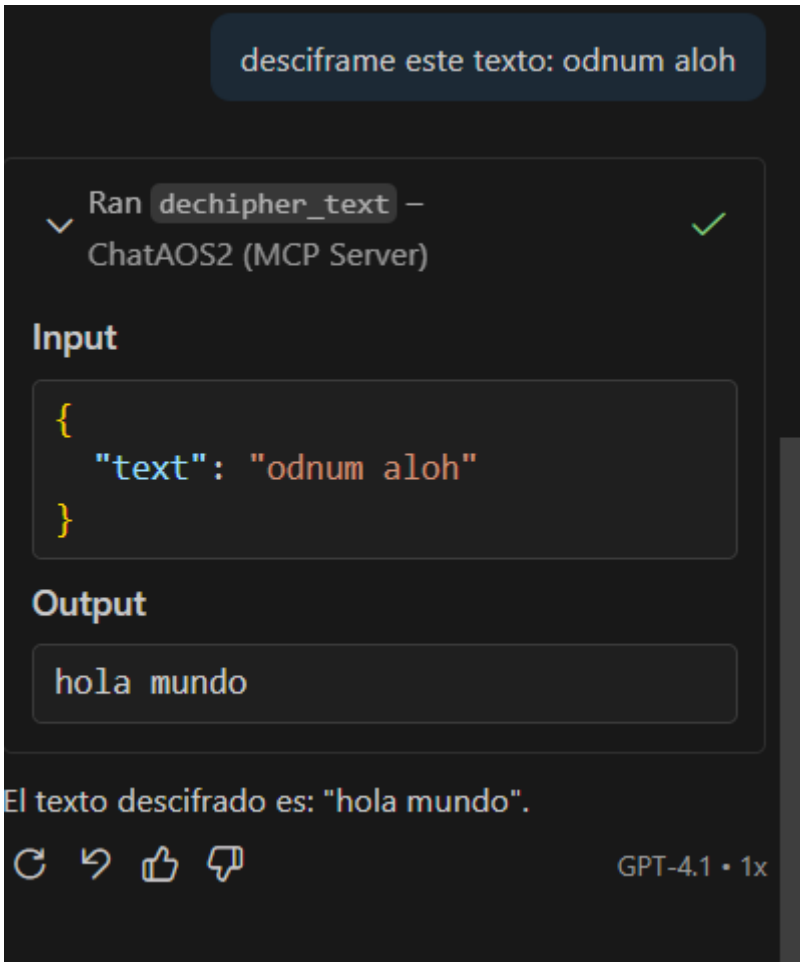

Saludame. me llamo ruben
Ran greet_user –
ChatAOS (MCP Server)
Input
{
"username": "ruben"
}
Output
Hola, ruben! Bienvenido al servidor FastMCP.
{
"result": "Hola, ruben! Bienvenido al servidor FastMCP."
}
¡Hola, ruben! Bienvenido al servidor FastMCP. ¿En qué puedo ayudarte hoy?
GPT-4.1 • 1x

Servidor 2

En el ejemplo previo, la aplicación del tool era bastante simple y la IA tampoco necesitaba gran cosa para cumplirla. En este caso, hemos creado un mensaje cifrado dándole la vuelta al orden de las letras de un texto tal que hola = aloh.

```python
from fastmcp import FastMCP

server = FastMCP("ChatAOS")

@server.tool()
def dechipher_text (text) :
    return text[::-1]

if __name__ == "__main__":
    server.run()
```

Esto levanta una tool que permite a la IA descifrar nuestro mensaje codificado. De nuevo, este ejemplo es bastante simple, pero con un cifrado más estricto, seguramente le sería de utilidad a tu IA.



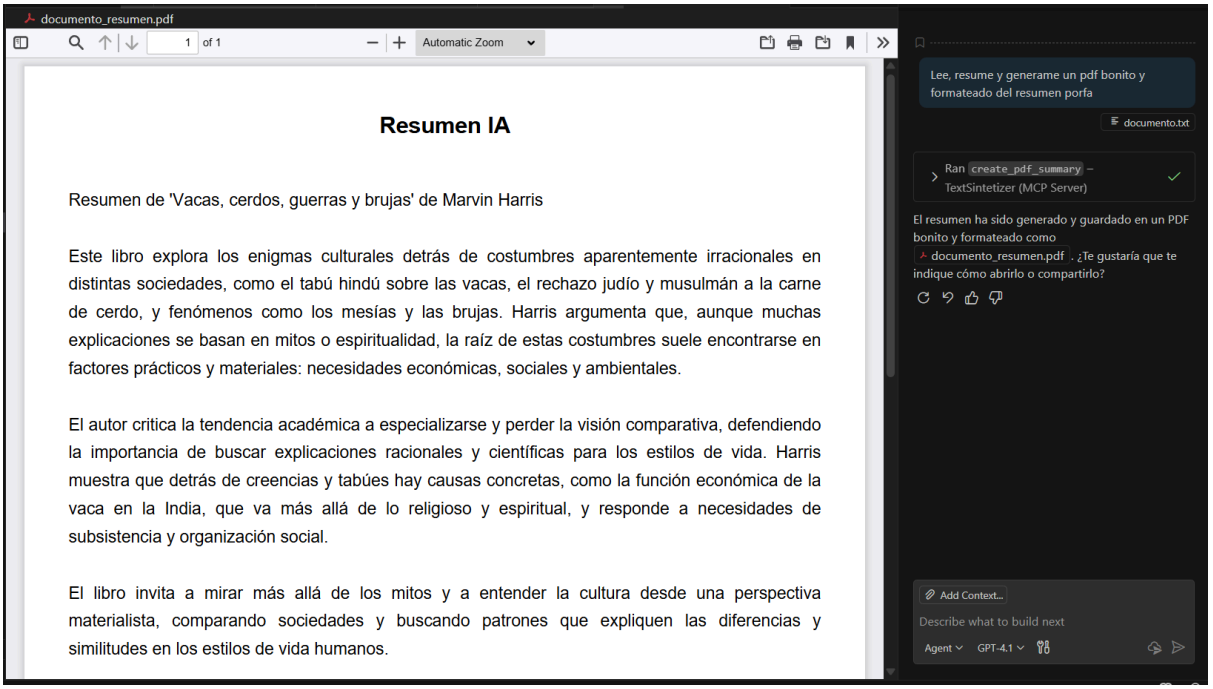
Servidor 3

Gemini no posee capacidad de generar artefactos pdf. Sin embargo, si ponemos un tool a su disposición, sí que pasa a ser capaz de generar pdf usando nuestro tool específico.

```
from fastmcp import FastMCP
from fpdf import FPDF
import os
import pypdf
from docx import Document
import sys

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
server = FastMCP("TextSintetizer")

def get_safe_path(filename_or_path: str) -> str:
    clean_filename = os.path.basename(filename_or_path)
    return os.path.join(BASE_DIR, clean_filename)

@server.tool()
def read_local_file(filepath: str) -> str:
    safe_path = get_safe_path(filepath)
    sys.stderr.write(f"[DEBUG] Leyendo archivo: {safe_path}\n")

    if not os.path.exists(safe_path):
        return f"Error: No encuentro el archivo
'{os.path.basename(safe_path)}' en /app."

    _, ext = os.path.splitext(safe_path)
    ext = ext.lower()

    try:
        if ext == '.pdf':
            reader = pypdf.PdfReader(safe_path)
            text = []
            for page in reader.pages:
                extracted = page.extract_text()
                if extracted:
                    text.append(extracted)
            return "\n".join(text)
        elif ext == '.docx':
            doc = Document(safe_path)
```

```
            return "\n".join([para.text for para in doc.paragraphs])
        else:
            with open(safe_path, 'r', encoding='utf-8') as f:
                return f.read()
    except Exception as e:
        return f"Error leyendo: {str(e)}"

@server.tool()
def create_pdf_summary(summary_text: str, filename: str = "resumen.pdf") ->
str:
    safe_path = get_safe_path(filename)
    sys.stderr.write(f"[DEBUG] Generando PDF en: {safe_path}\n")

    try:
        pdf = FPDF()
        pdf.add_page()
        pdf.set_font("Arial", 'B', 16)
        pdf.cell(0, 10, "Resumen IA", ln=True, align='C')
        pdf.ln(10)
        pdf.set_font("Arial", size=12)

        replacements = {'\u2018': "'", '\u2019': "'", '\u201c': '"', '\u201d':
'"', '–': '-'}
        for k, v in replacements.items():
            summary_text = summary_text.replace(k, v)

        clean_text = summary_text.encode('latin-1',
'replace').decode('latin-1')
        pdf.multi_cell(0, 7, clean_text)
        pdf.output(safe_path)

        return f"ÉXITO: PDF guardado como '{os.path.basename(safe_path)}'"
    except Exception as e:
        return f"Error: {str(e)}"

if __name__ == "__main__":
    server.run()
```

Posee dos tools. uno para leer ficheros, cosa que Gemini ya puede hacer, y otro para crear archivos pdf, lo cual Gemini por sí solo no es capaz de hacer. De esta forma, podemos hacer que un LLM realice operaciones que, por defecto, no podría realizar.



Servidor 4

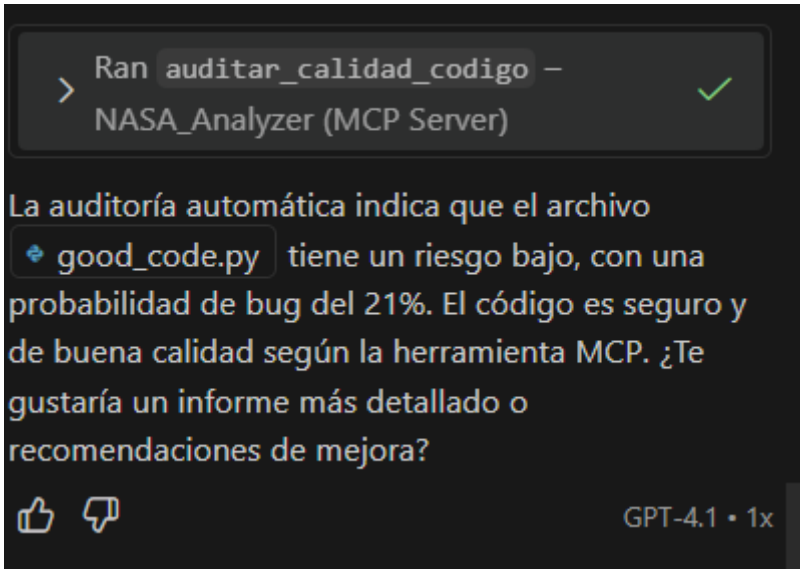
En este ejemplo, entrenamos una IA específicamente pensada para analizar código y ver si es bueno o malo. Una vez está entrenada, ofrecemos en un tool mcp una forma de analizar el código, de tal forma que Gemini solo tenga que leerlo y procesar la salida de la herramienta, evitando que se invente errores, o añada más features al código que no necesita.

```python
from fastmcp import FastMCP
import os
import sys

# Solo importamos lo ligero al principio
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
MODEL_FILE = os.path.join(BASE_DIR, "modelo_nasa_full.pkl")

server = FastMCP("NASA-Quality-Auditor")

@server.tool()
def auditar_calidad_codigo(ruta_archivo: str) -> str:
    """
    Analiza un archivo local. Carga el modelo bajo demanda.
    """
    # --- IMPORTS LAZY (Para que el servidor arranque rápido) ---
    import joblib
```

```
    import pandas as pd
    import radon.raw as raw
    import radon.complexity as cc
    from radon.metrics import h_visit

    # 1. Validación de rutas
    clean_filename = os.path.basename(ruta_archivo)
    safe_path = os.path.join(BASE_DIR, clean_filename)

    if not os.path.exists(safe_path):
        return f"Error: No encuentro '{clean_filename}' en la carpeta."

    if not os.path.exists(MODEL_FILE):
        return "Error CRÍTICO: No encuentro modelo_nasa_full.pkl"

    try:
        # 2. Carga del modelo (Solo ocurre cuando llamas a la tool)
        sys.stderr.write("[DEBUG] Cargando modelo NASA...\n")
        model_data = joblib.load(MODEL_FILE)
        model = model_data["model"]
        feature_names = model_data["features"]

        # 3. Lectura y Análisis
        with open(safe_path, 'r', encoding='utf-8') as f:
            code = f.read()

        r_analysis = raw.analyze(code)
        blocks = cc.cc_visit(code)
        total_cc = sum(b.complexity for b in blocks) if blocks else 0

        try:
            h_obj = h_visit(code)
            val_h1 = h_obj.total.h1
            val_effort = h_obj.total.effort
        except:
            val_h1, val_effort = 0, 0

        metricas = {
            "loc": r_analysis.loc,
            "complexity": total_cc,
            "h_uniq_op": val_h1,
            "comments": r_analysis.comments,
```

```
            "h_effort": val_effort
        }

        input_row = {feat: metricas.get(feat, 0) for feat in feature_names}
        df_input = pd.DataFrame([input_row])

        prob = model.predict_proba(df_input)[0][1]
        porcentaje = round(prob * 100, 2)

        riesgo = "CRÍTICO" if porcentaje > 75 else "MEDIO" if porcentaje > 40
else "BAJO"

        return f"AUDITORIA COMPLETA: Riesgo {riesgo} ({porcentaje}% bug
probability)."

    except Exception as e:
        return f"Error interno: {str(e)}"

if __name__ == "__main__":
    server.run()
```

De esta forma, evitamos que el modelo tenga que inventarse una respuesta y le dejamos que realice lo que mejor se le da: procesar una entrada de texto.



Ejercicio

Para poder practicar, haremos un ejercicio. Crearemos un servidor MCP que permita a la IA jugar a adivinar el número con el usuario:

- El servidor MCP le dará un número aleatorio a la IA y el usuario tratará de adivinarlo, mientras la IA usará las herramientas MCP para indicar al usuario si el número real es mayor o menor que el número adivinado.

Conclusiones

FastMCP es una solución estandarizada para conectar modelos de IA con software real, eliminando la complejidad de la implementación tradicional.

Los decoradores permiten convertir funciones de Python en herramientas operables por la IA sin necesidad de escribir rutas ni endpoints manuales.

Podemos integrarla con ecosistemas existentes mediante especificaciones OpenAPI, lo que permite generar herramientas automáticamente a partir de servicios REST ya existentes.

Bibliografía y recursos

[1] FastMCP Team, "Getting Started: Welcome," *FastMCP Documentation*. [En línea]. Disponible en: https://gofastmcp.com/getting-started/welcome. [Accedido: 09-dic-2025].

[2] Vunda.ai, "How MCP actually works and why FastMCP is the easiest way to use it," *Vunda.ai Blog*. [En línea]. Disponible en: https://www.vunda.ai/blog/fast-mcp-deep-dive. [Accedido: 09-dic-2025].

[3] Sreeni (sreeni5018), "Bridging the Gap Between LLMs and Enterprise APIs Using FastMCP," *Dev.to*, 2024. [En línea]. Disponible en: https://dev.to/sreeni5018/bridging-the-gap-between-llms-and-enterprise-apis-using-fastmcp-how-to-auto-generate-ai-tools-from-4cf9. [Accedido: 09-dic-2025].

[4] Birend17, "Getting Started with FastMCP: Building an SSE Server and Client," *Medium*, 2024. [En línea]. Disponible en: https://medium.com/@birend17/getting-started-with-fastmcp-building-an-sse-server-and-client-187bacc6aa02. [Accedido: 09-dic-2025].

[5] Apidog Team, "A Beginner's Guide to Use FastMCP," *Apidog Blog*, 2024. [En línea]. Disponible en: https://apidog.com/blog/fastmcp/. [Accedido: 09-dic-2025].