EDICIÓN #019

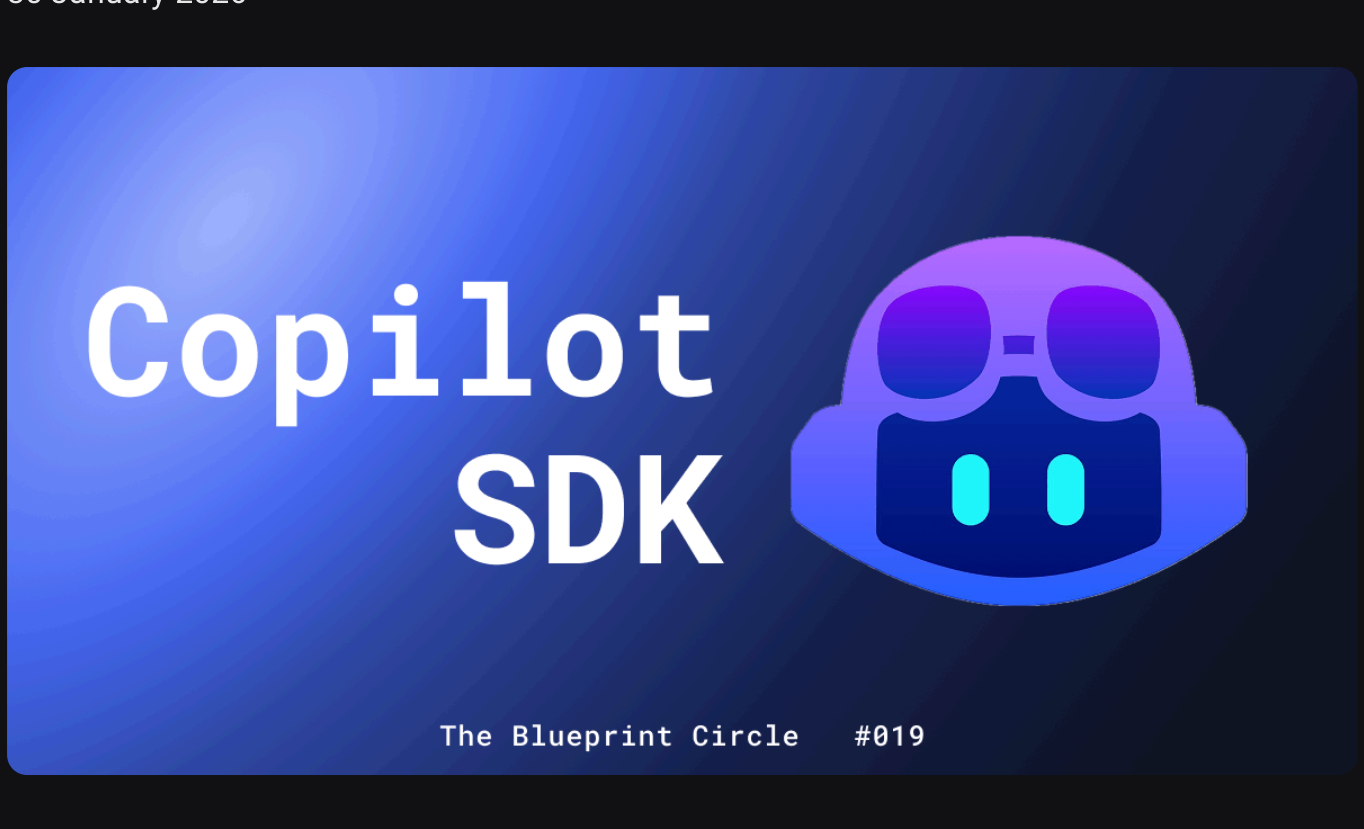
Guía completa de GitHub Copilot SDK: Crea agentes de IA en Python

30 de Enero de 2026

Guía completa de GitHub Copilot SDK: Crea agentes de IA en Python

Aprende a crear tu primer agente de IA capaz de ejecutar herramientas y comandos de sistema usando el SDK oficial de GitHub Copilot y Python.

30 January 2026



El otro día publicaba un post que tuvo muy buen recibimiento sobre patrones de diseño de agentes de IA y a raíz de él recibí algún mail preguntándome cómo empezar a crear agentes desde 0. Esa misma semana GitHub anunciaba en su blog el lanzamiento de GitHub Copilot SDK.

Los que os hayáis puesto a intentar **construir un agente de IA desde cero** os habréis dado cuenta de la cantidad de cosas que hay que montar antes de llegar a un agente sencillo sobre el que construir: tienes que gestionar el historial de la conversación, orquestar las llamadas a herramientas, manejar los reintentos si el modelo alucina con el JSON, autenticación... Hasta ahora, si querías integrar capacidades agénticas en tu aplicación, tenías que montarte tu propia plataforma o pelearte con frameworks complejos aunque también hay soluciones excelentes como la Google, Anthropic o de la que os vengo a hablar hoy.

GitHub ha lanzado, casi por sorpresa, el **GitHub Copilot SDK** en technical preview y lo que han hecho es darnos las posibilidad de **crear nuestros propios agentes** usando el motor que mueve Copilot.

Básicamente, ahora puedes **coger el runtime agéntico que usa GitHub en su CLI y en sus productos, y meterlo directamente en tus aplicaciones** en Python, Node.js, Go o .NET. Ya no tienes que construir el orquestador, ahora GitHub te lo da hecho para que tú solo te preocupes de definir las herramientas y la lógica general de lo que quieras construir.

He estado todo el fin de semana profundizando en el SDK y he creado una herramienta para limpiar mi Mac automáticamente basada en agentes. Te cuento todo lo que necesitas saber para empezar a usarlo hoy mismo.

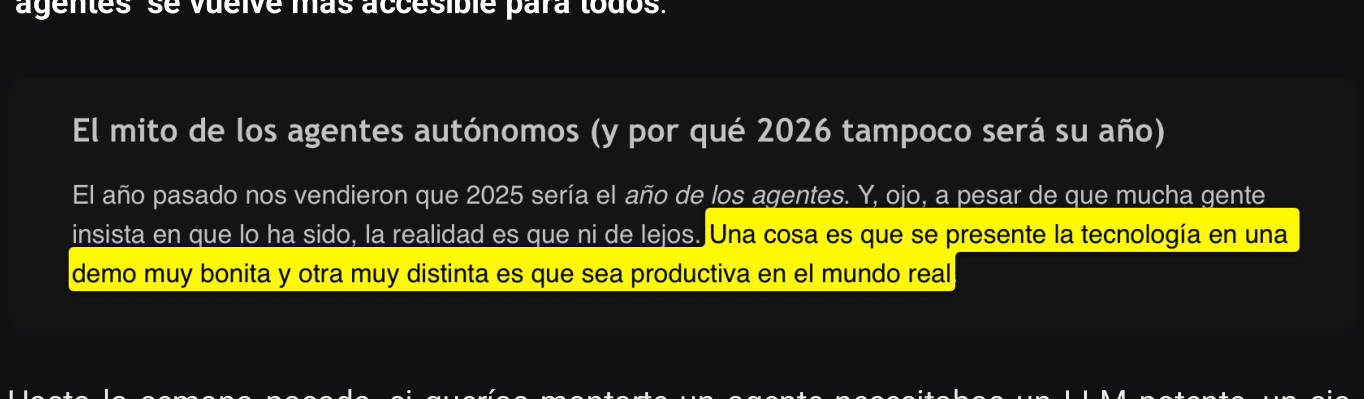
Resumen ejecutivo (TL;DR)

El nuevo SDK de GitHub Copilot SDK nos permite **integrar el motor agéntico de Copilot directamente en aplicaciones propias usando Python, TypeScript, Go o .NET**. Esto elimina la necesidad de construir orquestadores complejos desde cero, ya que el SDK gestiona la planificación, el uso de herramientas, el historial de contexto y la ejecución de comandos.

- **Motor Agéntico como Servicio:** Accedes al mismo runtime que usa GitHub Copilot CLI, delegando la gestión del contexto y la toma de decisiones al SDK.
- **Soporte Multi-lenguaje:** Disponibles librerías oficiales para los lenguajes más usados en backend y scripting.
- **Herramientas Personalizadas:** Puedes definir funciones en tu código (usando Pydantic en Python) que el agente puede invocar para interactuar con tu sistema o APIs externas.
- **Integración MCP:** Soporte nativo para el Model Context Protocol, permitiendo conectar servidores de contexto existentes (como acceso a ficheros o bases de datos) con una configuración mínima.
- **Streaming y Eventos:** Control sobre la respuesta del modelo, permitiendo interfaces en tiempo real y feedback visual del proceso de razonamiento del agente.

¿Qué es el GitHub Copilot SDK y qué puedo construir?

El 31 de diciembre de 2025 compartía contigo mi visión para este año 2026 en cuanto a lo que la IA respecta. Aquí te comentaba que 2025 no fue el año de los agentes y que 2026 tampoco lo sería pero que nos iríamos acercando. Hace un par de semanas Anthropic anunciaba Claude CoWork, la semana pasada GitHub anunciaba este SDK y poco a poco **la capacidad de crear 'agentes' se vuelve más accesible para todos**.



El mito de los agentes autónomos (y por qué 2026 tampoco será su año)

El año pasado nos vendieron que 2025 sería el *año de los agentes*. Y, ojo, a pesar de que mucha gente insista en que lo ha sido, la realidad es que ni de lejos. Una cosa es que se presente la tecnología en una demo muy bonita y otra muy distinta es que sea productiva en el mundo real

Hasta la semana pasada, si querías montarte un agente necesitabas un LLM potente, un sistema propio para limpiar y parsear sus respuestas, un bucle infinito que detectara si el modelo quería usar una herramienta, ejecutarla, devolver el resultado y volver a empezar. Y por si fuera poco, te tocaba pelearte con **la gestión de tokens y la ventana de contexto** para no arruinarte o que el modelo se olvidara de lo que le dijiste. (Si quieres profundizar en ventanas de contexto, gestión eficiente de tokens y context rot revisa el post de la semana pasada).

Lo que GitHub ha puesto encima de la mesa con este SDK es básicamente una abstracción de todo esto. Imagínatelo como una **capa intermedia programable** que te lo da todo hecho. Tú instancias un cliente, abres una sesión y le dices: "Oye, aquí tienes estas 5 herramientas que son funciones de mi código. El usuario quiere hacer X. Búscate la vida". Y **el SDK se encarga de hablar con el backend de Copilot**, decidir qué herramientas llamar, ejecutarlas y darte la respuesta final mascadita.

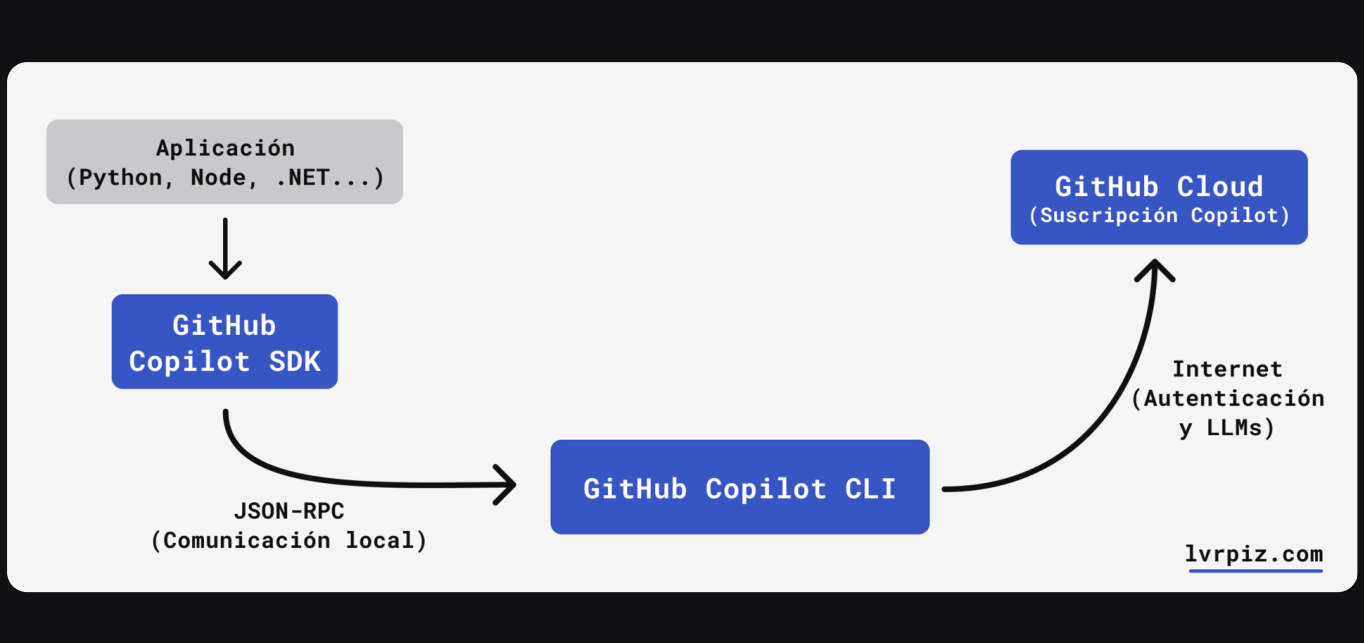


Diagrama de arquitectura GitHub Copilot SDK

Es importante **entender la arquitectura que hay debajo**. Esto funciona sobre la GitHub Copilot CLI. El SDK se comunica con el binario de la CLI (que actúa como servidor local) mediante JSON-RPC. Esto es muy interesante porque significa que **toda la autenticación y la gestión de modelos la heredas directamente de tu suscripción de GitHub Copilot**. Se acabó el andar gestionando claves de API de OpenAI o preocupándote por si quemas la tarjeta de crédito a mitad de mes, ya que usa la cuota que ya estás pagando en tu cuenta de GitHub.

Cómo crear tu primer agente con Python y GitHubCopilot SDK

Aunque el **SDK está disponible para varios lenguajes**, voy a usar **Python** para los ejemplos porque creo que es donde la mayoría nos sentimos cómodos para este tipo de scripts de automatización, y además es el lenguaje que he usado para el proyecto que os enseñaré luego.

Lo primero es lo primero y es que necesitas **tener instalada la GitHub Copilot CLI** y estar autenticado. Si no lo tienes, toca pasarse por la documentación oficial e instalarlo. Si lo vas a instalar con **npm** te sirve este comando para Linux, Windows y macOS, en la guía sino tienes otras opciones para hacerlo con brew o winget.

```
npm install -g @github/copilot
```

Una vez lo tengas, comprueba que funciona escribiendo `copilot --version` en tu terminal. Para el proyecto, crea un entorno virtual (siempre, por favor) e instala el paquete del SDK:

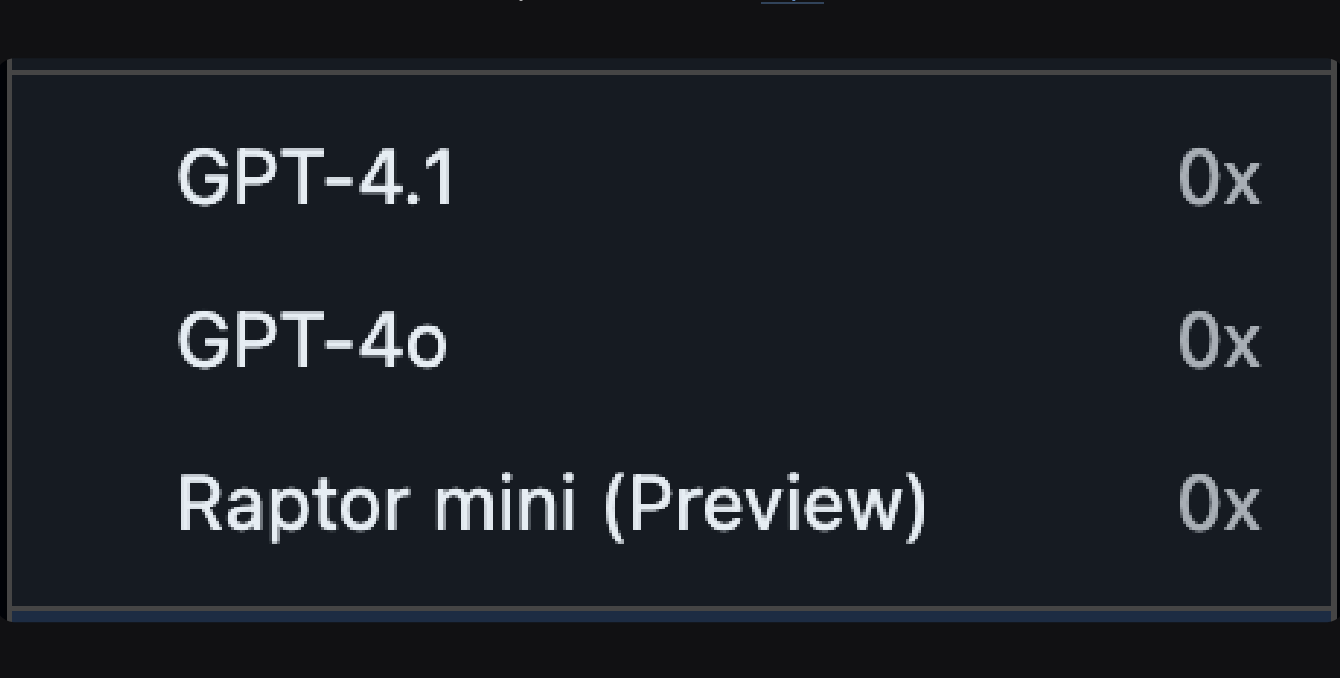
```
pip install github-copilot-sdk
```

Lo primero, vamos a crear un agente sencillo con 4 cosas básicas. Fíjate en la sencillez del código. No hay llamadas a endpoints REST complejos, solo objetos y métodos asíncronos.

```
import asyncio

from copilot import CopilotClient

async def main():

# Iniciamos el cliente. Esto arranca la CLI en modo servidor por debajo.

client = CopilotClient()

await client.start()

# Creamos una sesión (aquí podríamos definir herramientas, pero vamos a lo bás
session = await client.create_session({"model": "gpt-4.1"})

# Enviamos el prompt y esperamos.
response = await session.send_and_wait({"prompt": "Explícame qué es el Copilot

print(response.data.content)

await client.stop()

asyncio.run(main())
```

Si ejecutas esto, verás que tarda un poco y te escupe la respuesta de golpe. ¿Por qué? Porque `send_and_wait` hace exactamente lo que dice: espera a que termine todo el proceso. Pero en el mundo real, y sobre todo con LLMs, queremos ver qué está pasando, es decir, queremos streaming.

Un pequeño **consejo**: Usad GPT-4.1 o GPT-4o para todas estas pruebas ya que no os consumirán Tokens de la suscripción. Además, son modelos lo suficientemente buenos para la mayoría de los casos y siempre es algo que puedes cambiar después. Si queréis comprobar cuántos Tokens lleváis consumidos este mes lo podéis ver desde aquí.



GPT-4.1	0x
GPT-4o	0x
Raptor mini (Preview)	0x

Modelos de uso gratuito con la suscripción a GitHub Copilot

¿Cómo implementar Streaming y Eventos con GitHub Copilot SDK?

El SDK maneja el **streaming de mensajes a través de eventos**. En lugar de esperar pasivamente, **nos suscribimos a lo que va ocurriendo en la sesión**. Esto es crucial si estás construyendo una CLI interactiva o una interfaz web, porque el usuario necesita ver que algo se mueve.

El cambio en el código es mínimo pero el resultado es mucho más profesional. Activamos `streaming: True` en la configuración de la sesión y definimos un manejador de eventos:

```
import asyncio

from copilot import CopilotClient

async def main():

# Iniciamos el cliente. Esto arranca la CLI en modo servidor por debajo.

client = CopilotClient()

await client.start()

# Creamos una sesión. Aquí podríamos definir herramientas, pero vamos a lo bás
# Fíjate que podemos elegir el modelo, en este caso GPT-4.1 y activar el strea
session = await client.create_session({"model": "gpt-4.1", "streaming": True})

# Definimos la función para manejar eventos
def on_event(event):
# Filtramos solo los eventos relevantes (sino habría muchos otros eventos
if event.type.value == "assistant.message_delta" and event.data.delta_cont
print(event.data.delta_content, end="", flush=True)
elif event.type.value == "assistant.reasoning_delta" and event.data.delta_
print(f"[PENSANDO: {event.data.delta_content}]", end="", flush=True)

# Vinculamos la función de eventos a la sesión
session.on(on_event)

# Enviamos el prompt y esperamos.
response = await session.send_and_wait({"prompt": "Explícame qué es el Copilot

print(response.data.content)

await client.stop()

asyncio.run(main())
```

En el ejemplo previo sin eventos el código espera pasivamente a que el modelo complete la respuesta y luego imprime le resultado, no hay retroalimentación en tiempo real para el usuario mientras el modelo procesa el prompt. En este nuevo ejemplo activamos `“streaming”: True` y definimos un manejador de eventos de tal modo que **podemos procesar mensajes parciales** y el **usuario puede ver cómo el modelo genera la respuesta progresivamente**.

Además, poder acceder al `reasoning_delta` significa que si en el futuro conectamos modelos como o1 o sus sucesores, podremos **mostrarle al usuario "qué está pensando"** el agente antes de actuar.

¿Cómo añadir Herramientas Personalizadas (Tools) usando el GitHub Copilot SDK?

Aquí es donde la cosa se pone muy interesante y vemos gran parte del poder del SDK. Un agente que solo habla es un chatbot. **Un agente que puede ejecutar código es ya algo más serio como lo que vemos con Claude Cowork** o Claude Code.

El SDK de Python utiliza Pydantic para la definición de esquemas, lo cual es excelente ya que si has utilizado FastAPI es bastante similar. No tienes que escribir JSON Schemas a mano propensos a errores. **Defines una clase, le pones tipos, añades descripciones y el SDK se encarga de traducirlo para que el LLM lo entienda**.

Imagínate que queremos que nuestro agente pueda consultar el estado de un servicio o, en un caso más doméstico, el tiempo.

```
import asyncio
from pydantic import BaseModel, Field
from copilot import CopilotClient, define_tool

# Definimos la estructura de los argumentos que necesita nuestra herramienta
class WeatherParams(BaseModel):
city: str = Field(description="El nombre de la ciudad para consultar el tiempo

# Creamos la función y la decoramos
@define_tool(description="Obtiene el tiempo actual de una ciudad específica")
async def get_weather(params: WeatherParams) -> str:
# Aquí iría tu lógica real, llamada a una API externa, etc.
# Simulamos para el ejemplo
return f"En {params.city} hace 25 grados y está soleado."

async def main():

# Iniciamos el cliente. Esto arranca la CLI en modo servidor por debajo.
client = CopilotClient()
await client.start()

# Creamos una sesión con herramientas personalizadas
session = await client.create_session({
"model": "gpt-4.1",
"streaming": True,
"tools": [get_weather],  # Añadimos la herramienta
})

# Definimos la función para manejar eventos
def on_event(event):
# Filtramos solo los eventos relevantes
if event.type.value == "assistant.message_delta" and event.data.delta_cont
print(event.data.delta_content, end="", flush=True)
elif event.type.value == "assistant.reasoning_delta" and event.data.delta_
print(f"[PENSANDO: {event.data.delta_content}]", end="", flush=True)

# Vinculamos la función de eventos a la sesión
session.on(on_event)

# Enviamos el prompt y esperamos.
response = await session.send_and_wait({
"prompt": "¿Qué tiempo hace en Valladolid?"
})

print(response.data.content)

await client.stop()

asyncio.run(main())
```

Cuando tú le preguntes al agente "¿Qué tiempo hace en Valladolid?", pasará lo siguiente:

1. El agente analiza tu intención.
2. Ve que tiene una herramienta `get_weather` que encaja.
3. El SDK pausa la generación, ejecuta tu función `get_weather` con el parámetro "Valladolid".
4. El SDK inyecta el resultado ("En Valladolid hace 3 grados...") de vuelta al contexto del modelo.
5. El modelo genera la respuesta final en lenguaje natural: "Ahora mismo en Valladolid tenéis 3 grados y niebla".

Ahora toda esa transición de ida y vuelta, que antes te costaba 100 líneas de código gestionar, ahora es automático.

Caso real: Creando un agente para liberar y limpiar mi mac

La herramienta que uso para limpiar mi mac y liberar espacio es Mole entonces pensé en crear un agente real que me ayude a gestionar el espacio en disco de mi Mac pero empleando por debajo todo lo que hemos visto usando el SDK de GitHub Copilot y varias Tools.

El resultado es un proyecto que he llamado **AgentCleansMac** que por cierto es de código abierto, os animo a probarlo. La idea es sencilla: un script de **Python** que utiliza el **Copilot SDK** para analizar mi sistema, buscar carpetas caché gigantes (como las de Xcode, npm, o Docker) y **sugerirme qué borrar**.

```
MAC CLEANER
AI-Powered Mac Cleaning Assistant
Commands
Command     Description
scan        Full system scan - find all cleanable items
clean       Interactive cleanup - choose what to remove
quick       Instant safe cleanup - caches and logs only

status      Show disk space usage
large       Find large files not used recently
purge       Find project artifacts (node_modules, etc.)
uninstall   Completely remove an app and its leftovers

help        Show this menu
clear       Clear screen
exit        Quit

Hello! I'm Mac Cleaner.

I can help you free up disk space by cleaning:
  • System caches, logs, and temp files
  • Browser caches (Chrome, Firefox, Safari)
  • Developer tools (Xcode, npm, pip, Homebrew, Docker)
  • iOS backups, Time Machine snapshots
  • And much more...

Type scan to see what's using space, or just ask me anything.

> You (): status

⚡ report_intent...
⚡ Checking disk...
                         Disk Usage
   Used: 129Gi (63%)    Free: 79Gi    Total: 228Gi

Disk status: 79Gi free of 228Gi total (63% used).

> You (): scan

⚡ report_intent...
⚡ Scanning system...
                 Scan Results — 0B can be cleaned
Code Editors
└── ✓ VS Code Cached Data (10.5 MB)
Docker
└── ⚠ Docker Data (4.5 GB) — manual only
Cleanable items found (by category):

- Code Editors:
  - VS Code Cached Data: 10.5 MB (✓ Safe)
- Docker:
  - Docker Data: 4.5 GB (⚠ High, manual only, use: docker system prune -a -f --volumes)

Total cleanable: 4.5 GB across 2 items.

> You (): █
```

Captura de pantalla de AgentCleansMac

Podéis ver el código completo en el repositorio, pero en resumen el agente, por su cuenta:

1. Llama a herramientas para ver espacio total. Ve que me quedan 5GB por ejemplo.
2. Piensa (y lo veo en el log): *El usuario tiene poco espacio. Voy a mirar las carpetas de caché habituales.*
3. Llama a `analyze_directory` en `~/Library/Caches`.
4. Detecta que Hugging Face por ejemplo tienen 20GB de caché.
5. Me responde: "He visto que Hugging Face está ocupando 20GB de caché. ¿Quieres que lo borre?".

Si le digo que sí, llama a la herramienta `safe_delete`.

Este nivel de interacción, donde el modelo entiende la semántica de mis archivos y ejecuta comandos de sistema bajo supervisión, es lo que **diferencia a un script básico de un agente de IA**. Y con el SDK, montar todo el esqueleto me llevó menos de una hora.

Capacidades Avanzadas: MCP y Modelos

Hay dos cosas más que quiero comentar antes de cerrar y que me parecen diferenciales respecto a otras soluciones como LangChain.

La primera es la integración con **MCP (Model Context Protocol)**. Ya os hablé de MCP hace unas semanas. Es el estándar para conectar datos a LLMs. El **SDK de Copilot soporta esto de forma nativa**. Puedes decirle al cliente: "Oye, conéctate también a este servidor MCP que tengo corriendo localmente y que tiene acceso a mi base de datos de Notion".

```
import asyncio
from pydantic import BaseModel, Field
from copilot import CopilotClient, define_tool

# Definimos la estructura de los argumentos que necesita nuestra herramienta
class WeatherParams(BaseModel):
city: str = Field(description="El nombre de la ciudad para consultar el tiempo

# Creamos la función y la decoramos
@define_tool(description="Obtiene el tiempo actual de una ciudad específica")
async def get_weather(params: WeatherParams) -> str:
# Aquí iría tu lógica real, llamada a una API externa, etc.
# Simulamos para el ejemplo
return f"En {params.city} hace 25 grados y está soleado."

async def main():

# Iniciamos el cliente. Esto arranca la CLI en modo servidor por debajo.
client = CopilotClient()
await client.start()

# Definimos el modelo y el servidor MCP
model_name = "gpt-4.1"  # Cambia este valor para usar otro modelo
mcp_servers = {
"sqlite_db": {
"type": "http",
"url": "http://localhost:8080/mcp"  # Servidor MCP local
}
}

# Creamos una sesión con herramientas personalizadas y MCP
session = await client.create_session({
"model": model_name,
"streaming": True,
"tools": [get_weather],  # Añadimos la herramienta
"mcp_servers": mcp_servers,  # Conectamos el servidor MCP
})

# Definimos la función para manejar eventos
def on_event(event):
# Filtramos solo los eventos relevantes
if event.type.value == "assistant.message_delta" and event.data.delta_cont
print(event.data.delta_content, end="", flush=True)
elif event.type.value == "assistant.reasoning_delta" and event.data.delta_
print(f"[PENSANDO: {event.data.delta_content}]", end="", flush=True)

# Vinculamos la función de eventos a la sesión
session.on(on_event)

# Enviamos el prompt y esperamos.
response = await session.send_and_wait({
"prompt": "¿Qué tiempo hace en Valladolid?"
})

print(response.data.content)

await client.stop()

asyncio.run(main())
```

Automáticamente, **todas las herramientas que exponga ese servidor MCP pasan a estar disponibles para el agente**. Esto permite una arquitectura modular donde tienes tu agente cerebro en Python, y le vas enchufando módulos de capacidades vía MCP sin tener que tocar el código del agente.

La segunda es la flexibilidad de usar el modelo que quieras. El SDK está preparado para cambiar de modelo fácilmente. En el código simplemente cambias el string `"model": "gpt-4.1"` o `"claude-3.5-sonnet"` y listo. Lo ideal es que en función de la lógica del agente quizás para unas tareas puedas usar modelos más eficientes y baratos y para tareas más complejas usar modelos razonadores más avanzados.

Conclusión: Un paso más cerca de agentes de verdad

La llegada de estos SDKs en mi opinión marca un punto de inflexión porque hasta ahora, **el 80% del tiempo de desarrollo de una aplicación con IA se iba en montar la infraestructura para que el LLM pudiera simplemente hacer cosas**. Con esto, ese porcentaje se invierte. Ahora el 80% del tiempo lo vas a dedicar a lo que importa que es diseñar buenas herramientas, asegurar que sean seguras y definir la lógica de negocio.

SDKs como el de GitHub nos convierte en arquitectos de agentes. Ya no tenemos que ser expertos en *prompt engineering* de bajo nivel o en gestión de colas de mensajes para tener un agente funcional. Simplemente definimos interfaces (tools) y dejamos que el motor de Copilot haga el resto.

Mi recomendación es clara: si tienes scripts de mantenimiento, herramientas CLI internas en tu empresa o procesos manuales que impliquen mirar datos y tomar decisiones, dales una vuelta con este SDK. Es muy probable que puedas convertir un script rígido en un agente flexible en una tarde, la pregunta ahora es si realmente merece la pena hacerlo. **En muchos casos esta capacidad agéntica no es más que una interfaz más con la que interactuar con nuestras aplicaciones**.

Recordaros que este SDk está en *technical preview*. Seguro que vamos a ver cambios, mejoras y soporte para más modelos en los próximos meses. Pero la base es sólida y, sobre todo, tremendamente fácil de usar.

Referencias técnicas

1. Documentación Oficial del GitHub Copilot SDK - Repositorio principal con guías y ejemplos.
2. Anuncio oficial en el Blog de GitHub - Visión general y casos de uso presentados por el equipo de producto.
3. Repositorio de AgentCleansMac - El código fuente del agente de limpieza que he creado como ejemplo práctico. Proyecto de código abierto para que puedas clonarlo, mejorarlo y que lo vayamos construyendo juntos.
4. Guía de Pydantic para definición de modelos - Fundamental para definir correctamente los esquemas de las herramientas en Python.
5. Model Context Protocol (MCP) - Especificación oficial del protocolo para conectar fuentes de datos externas.

Recurso de referencia. Disponible en los siguientes formatos:

PDF
Markdown
Explora el Blueprint con IA