VScode y GitHub Copilot

> Configuración y uso del agente de IA de GitHub

GitHub Copilot se ha convertido en el asistente de IA más popular para programadores. Y no es casualidad: su integración con **Visual Studio Code** es tan fluida que a veces olvidas que estás trabajando con una IA.

Pero, ¿sabes realmente cómo **configurarlo** para que te ayude de la manera más eficiente? ¿Conoces todas las opciones que tienes para personalizar su comportamiento?

En este artículo vamos a ver cómo **explotar al máximo** las capacidades de GitHub Copilot en VSCode, desde la configuración básica hasta técnicas avanzadas de **prompt y context engineering**.

Configuración básica de los ajuste de copilot

Lo primero es lo primero. Antes de empezar a usar GitHub Copilot, necesitas tener todo **correctamente configurado**.

Mi recomendación: **activa todo** y luego ajusta según tus preferencias. En Settings de VSCode, busca "GitHub Copilot" y verás varias opciones:

```
{
  "chat.agent.maxRequests": 100,
  "chat.tools.autoApprove": true,
  "github.copilot.advanced": {
    "inlineSuggest": true,
    "inlineSuggest.enable": true,
    "inlineSuggest.showSuggestionsAlways": true
  },
  "github.copilot.chat.codesearch.enabled": true,
  "github.copilot.chat.agent.thinkingTool": true,
  "github.copilot.chat.languageContext.[your-language].enabled": true,
  "github.copilot.enable": {
    "*": true,
    "inlineSuggest": true,
    "inlineSuggest.enable": true,
    "inlineSuggest.showSuggestionsAlways": true
  },
  "github.copilot.nextEditSuggestions.enabled": true,
}
```

Instrucciones

Aquí es donde **GitHub Copilot** se diferencia de otros asistentes. Te permite configurar **instrucciones personalizadas** que guían su comportamiento.

Las instrucciones se escriben en formato Markdown en ficheros `.md` en la carpeta `.github` del root de tu proyecto.

GitHub Copilot maneja las instrucciones de forma **jerárquica** a distintos niveles: organización, usuario o proyecto. Y esto siempre genera un poco de lio. Para empezar vamos a sacar de la ecuación a las instrucciones de organización, porque se activan en eso, una organización de GitHub y requieren licencias especiales.

Nos quedan dos: A nivel de usuario, es decir que sean válidas para todos tus proyectos, y las de repositorio, es decir que sean válidas para un proyecto concreto. Empecemos por estas, que son las más útiles.

Instrucciones de repositorio

Se crean en un fichero Markdown `.github/copilot-instructions.md` en el root de tu proyecto. Ojo al nombre de la carpeta es `.github` con el punto inicial. Lo que escribas en ese fichero se incluirá en cada petición de Copilot, asi que procura que sea lo más conciso y general posible. Por ejemplo añade:

- Tu **nivel de experiencia** como programador
- **Lenguajes preferidos** y frameworks
- **Estilo de código** que prefieres
- **Patrones de diseño** que sueles usar
- **Idioma de respuesta** preferido
- **Formato de salida** que necesitas

Cuando empiezas quizás sea suficiente con un párrafo que agrupe estas preferencias. Repito, ten en cuenta que se agrega a cada petición.

Este es un ejemplo de lo que podría contener un fichero de instrucciones (no lo tomes al pie de la letra, es sólo una idea, si quieres ver algo más profesional visita AIDDbot copilot instructions):

```
# Instrucciones para [Nombre del proyecto]

- Usa TypeScript estricto
- Estás en un sistema Windows 11
- Usa el terminal git bash para todos los comandos de consola
- Si no está disponible, usa el command-prompt de Windows
- Escribe código y documentación en inglés, pero habla con el usuario en su idioma
- Sigue el patrón Repository para acceso a datos
- Implementa tests unitarios para todas las funciones
- Documenta las APIs con JSDoc
- Usa ESLint y Prettier para formateo
- Evita el uso de `any`, usa tipos específicos
- Implementa manejo de errores con try-catch
- Usa constantes para valores mágicos
```

Recomendación práctica: mantén las instrucciones del repositorio cortas, precisas y orientadas a acciones. Evita instrucciones ambiguas o que dependan de recursos externos no accesibles por Copilot.

Instrucciones por contexto

GitHub Copilot también puede usar instrucciones específicas para cada contexto. Por ejemplo, si estás trabajando en un proyecto de Angular, o si te gusta usar un determinado ORM en Java.

Estas instrucciones se almacenan en la carpeta `.github/instructions` del root de tu proyecto en ficheros con este patrón de nombre: `[context].instructions.md` y una estructura interna con una cabecera en Front-matter que indica propósito y ficheros a los que aplica.

Por ejemplo, podríamos tener un fichero de instrucciones para Node 24 llamada `.github/instructions/node24.instructions.md`:

```
---
description: "Node 24 Framework Best Practices"
applyTo: "**/*.ts"
---
# Node 24 Framework Best Practices

## 🚫 AVOID These Dependencies

**Never install these when using modern Node.js:**

- _ts-node_ - Not needed with native TypeScript support
- _nodemon_ - Use `node --watch` instead
- _jest_ or any other testing library (Use native `node --test` instead)
- _axios_ or any HTTP client library (use native `node:http` or `node:https`)
- Any TypeScript compilation loaders

## Import Instructions

- Use the `.ts` suffix for TypeScript files.
- Use `type` for type imports.
```

Este contenido se agregaría como contexto cada vez que el agente de Copilot esté trabajando con ficheros de TypeScript.

> Nota: Estos ficheros también se puede guardar en la carpeta local de del perfil de usuario, y así se aplican a todos los proyectos. Pero, es más cómodo y práctico hacerlo custom para cada repositorio y copiar a mano de algún sitio accesible.

Para que te hagas una idea de su potencial, visita estos repositorios de GitHub:

- GitHub Awesome Copilot instructions
- AIDDbot instructions

Invocación manual de instrucciones

No todo es **sugerencias automáticas** en base al *glob* `applyTo` de los ficheros de instrucciones. GitHub Copilot también permite invocar ficheros de instrucciones dentro de un *prompt* para que se tengan en cuenta. A veces se usan como plantillas para generar documentación o simplemente contexto específico para una petición concreta.

```
Crea un documento de estructura de la aplicación siguiendo las instrucciones #file:instructions/project-structure.instructions.md y confirma el resultado en git según las instrucciones #file:git-commit.instructions.md
```

De esta forma queda explícito qué instrucciones seguir. Y, sí, también se puede hacer simplemente agregando el fichero o ficheros de instrucciones al contexto del chat, pero entonces solo aplica en ese momento y no es fácil reutilizarlo; Eso es justo lo que veremos a continuación.

Biblioteca de prompts

Hay cierto tipo de peticiones que hacemos una y otra vez. Tal es así, que Copilot incluye una serie de prompts predefinidos que se invoca como si fuesen comandos. Siguen un convenio de nombres que les identifica. Son los **slash-commands**.

Comandos Slash

Los comandos slash son atajos predefinidos que ejecutan acciones específicas:

Comandos básicos de gestión:

- `/clear`: Inicia una nueva sesión de chat
- `/help`: Referencia rápida y conceptos básicos de GitHub Copilot

Comandos de análisis y explicación:

- `/explain`: Explica el código seleccionado
- `/fix`: Corrige problemas en el código

Comandos de generación:

- `/tests`: Genera tests para el código
- `/new`: Crea un nuevo proyecto
- `/fixTestFailure`: Encuentra y corrige una prueba que falla

Para usar un comando slash, escribe `/` en la caja de prompt del chat, seguido del nombre del comando. La lista de comandos disponibles puede variar y ser aplicada con tus propios comandos.

Comandos personalizados

Copilot permite crear comandos personalizados usando archivos de prompt similares a las instrucciones de repositorio.

La diferencia es que se almacenan en la carpeta `.github/prompts` y su nombre debe ser `[command].prompt.md`. Y, como en el caso de las instrucciones, empiezan con un Front-matter.

Ejemplo de comando `/refactor` en el fichero `.github/prompts/refactor.prompt.md`:

```
---
description: "Refactorización de código"
---
# Refactorización de código

Refactoriza el código seleccionado siguiendo estos principios:
- Extrae funciones pequeñas y con una sola responsabilidad
- Usa nombres descriptivos para variables y funciones
- Elimina código duplicado
- Mantén la funcionalidad original
- Sigue las instrucciones de #file:clean-code.instructions.md
- Al terminar, confirma ejecutando el comando [/git-commit](.github/prompts/git-commit.prompt.md)
```

Como ves, los **archivos de prompt** pueden referenciar otros archivos del proyecto usando sintaxis Markdown estándar o la sintaxis específica `#file:folder/file-name.extension`. Esto permite proporcionar contexto adicional como especificaciones de API o documentación del producto.

Te resumo algunas de las variables de chat que puedes usar, tanto en prompts guardados en ficheros como en el prompt que escribes en el chat:

Variables de chat enriquecen cualquier modo:

- `#file`: Incluye el contenido del archivo especificado
- `#selection`: Incluye el texto seleccionado actualmente
- `#function`: Incluye la función o método actual

Participantes de chat actúan como expertos especializados:

- `@workspace`: Considera la estructura completa de tu proyecto
- `@vscode`: Ayuda específica con Visual Studio Code
- `@terminal`: Asistencia con comandos de terminal
- `@github`: Skills específicos de GitHub

De nuevo, te dejo un par de repositorios de ejemplo para que veas todo lo que se puede hacer:

- AIDDbot command prompts

- GitHub Awesome Copilot command prompts

Un último matiz con respecto a las diferencias entre instrucciones y prompts. Los comandos personalizados son siempre voluntarios, no se activan por contexto, así que no tiene `applyTo`. A cambio te permiten configurar, el modelo, las herramientas y hasta el modo de chat.

¿No sabes qué son los modos de chat? Sigue leyendo.

Modos de chat

VSCode ofrece diferentes **modos de chat** que se adaptan a distintas necesidades durante el desarrollo. Cada modo optimiza la experiencia para tipos específicos de tareas. De fábrica hay tres: Ask, Edit y Agent.

Modo Ask

Para **preguntas rápidas** y consultas. Es el más directo y conversacional. Ideal para conocer el código o aprender sobre algún uso específico.

Modo Edit

Especializado en **modificar código**. Te permite seleccionar código y pedir cambios específicos. Perfecto para correcciones rápidas, explicaciones de funciones específicas o sugerencias de mejora dentro de un fichero.

Modo Agent

El más potente. Actúa como un **asistente de desarrollo** que puede:

- Analizar todo tu workspace
- Crear, modificar y borrar archivos y carpetas
- Ejecutar múltiples tareas en secuencia
- Ejecutar comandos
- Usar herramientas MCP

Este modo es especialmente útil para encargarle a Copilot tareas reales y complejas. Es decir, este es el modo por defecto que usarás a diario... hasta que veas como crear el tuyo propio.

Modos de chat personalizados

Pues eso, que puedes crear nuevos modos de chat personalizados para tus necesidades. Es similar a la creación de instrucciones y comandos personalizados. Para ello debes crear un fichero `*.chatmode.md` en la carpeta `.github/chatmodes` del root de tu proyecto.

La estructura del Front-matter es la misma que para los comandos, pero su uso es más parecido al de las instrucciones. Me explico. El modo de chat, una vez seleccionado, agrega su contenido a cada prompt. Es como una extensión del fichero de instrucciones general. O como unas instrucciones con glob `*.*`, pero sin necesidad de explicitarlo.

Ejemplo de modo de chat `revisor` personalizado en el fichero `.github/chatmodes/revisor.chatmode.md`:

```
---
description: "Soy un revisor de código"
---

# Soy un revisor de código

- No puedo cambiar código
- No puedo agregar funcionalidades
- Seguiré las instrucciones de #file:clean-code.instructions.md
- Solamente haré un informe de errores y recomendaciones
- Será una lista priorizada para que copien y peguen en el chat
```

Para usarlo, simplemente selecciona el modo `revisor` de chat en el chat de Copilot y haz una petición como esta:

```
Revisa la clase #file:my-class.ts
```

De nuevo, los ejemplos más interesantes los puedes encontrar en los repositorios de GitHub:

- AIDDbot chat mode
- GitHub Awesome Copilot chatmodes

El potencial de estos modos aumenta cuando además les asignas un conjunto de herramientas que pueden usar. ¿Herramientas? ¿Qué herramientas?. Anda, sigue leyendo, que ya falta poco para que domines todo lo que puedes hacer con GitHub Copilot.

Herramientas y MCP

GitHub Copilot puede **extenderse** con herramientas externas usando el protocolo MCP (Model Context Protocol).

¿Qué es MCP?

El protocolo MCP permite a los modelos de IA **interactuar con herramientas externas en lenguaje natural**. Entre otras, puedes solicitar peticiones que incluyan acciones como estas:

- Ejecutar comandos en terminal
- Conectarse a APIs externas
- Acceder a bases de datos
- Usar herramientas de desarrollo específicas
- Consultar documentación de APIs
- Validar código contra reglas de negocio
- Integrar con herramientas de CI/CD

Por ejemplo, si instalas el MCP de GitHub, podrías pedirle cosas como:

- Crea un repositorio en mi cuenta de GitHub llamado "mi-mega-proyecto"
- Dame la lista de repositorios de la organización "mi-empresa"
- Dame la lista de issues del repositorio "mi-repo"
- Crea una issue en el repositorio "mi-repo" con el título "mi-issue"

Todo, como ves, en lenguaje natural y usando la herramienta de manera inteligente. Pero también puedes especificar la herramienta que quieres usar con un prompt.

```
- Crea un repositorio en mi cuenta de GitHub llamado "mi-mega-proyecto" usando
#GitHubIssues
```

Configuración de MCP

Lo primero es entender un par de conceptos básicos. Como dije, MCP es un protocolo creado por Anthropic para que los modelos de IA puedan usar herramientas externas. La implementación se hace a traves de servidores MCP, los cuales hay que instalar y configurar. Una vez hecho, deben iniciarse antes de usarlos. Recuerda, son servicios que se ejecutan en segundo plano. Una vez en funcionamiento ofrecerán una lista de herramientas, y esas herramientas serán las que usen los LLMs.

La manera más sencilla de instalar un servidor MCP es usar el market place de herramientas MCP.

Esto genera un fichero de configuración en la carpeta `.vscode/mcp.json` que puedes editar para configurar las herramientas que quieres usar.

Ejemplo de configuración para usar el MCP de GitHub:

```
{
  "servers": {
        "github": {
            "type": "http",
            "url": "https://api.githubcopilot.com/mcp/"
        }
    }
}
```

Normalmente, será necesario un proceso de credenciales y permisos que dependerá del servicio que uses. Puede requerir una API key o un token OAuth.

Conjuntos de herramientas

Cuando hayas instalado y lanzado unos cuantos servidores MCP verás que el número de herramientas disponibles es enorme. Copilot lo detecta y te lanza una advertencia si estás usando demasiadas (al final esto influye en su memoria de contexto).

La idea es que las actives según sea necesario. Esto puedes hacerlo desde la interfaz del chat o en el Front-matter de los ficheros de prompt y chat mode. En cualquier caso, se hace tedioso seleccionar una y otra vez ciertas herramientas. Créeme, tendrás muchísimas más herramientas que usar de las que te imaginas.

Afortunadamente, aparece el concepto de conjunto de herramientas que te permite agrupar herramientas en unidades lógicas, darles un nombre y activarlas todas a la vez.

Una particularidad de algunos servidores es que tras instalarse, te ofrecen sus propios comandos que ya incluyen sus herramientas. Este es un mundo en constante expansión.

Casos de uso avanzados

EL protocolo MCP te permite envolver cualquier servicio o incluso utilidades de terminal de forma que se ofrezca a clientes MCP (Copilot es un cliente MCP) para ser invocados en lenguaje natural.

Por ejemplo, puedes crear un servidor MCP para tu propio servicio o utilidad de terminal. Y luego usarlo en Copilot o desde un chat con un LLM. No me sorprenderá que el la generación de servidores MCP se convierta en una característica más de las aplicaciones que hagamos a partir de ahora.

Recuerda, el objetivo es manejar cualquier software mediante instrucciones en lenguaje natural, que después los agentes traduzcan a llamadas API, comandos CLI o acciones de usuario.

Conclusión

GitHub Copilot en VSCode es mucho más que un auto-completado inteligente. Es un **ecosistema completo** de desarrollo asistido por IA que puedes personalizar hasta el último detalle.

La clave está en la **progresión gradual**: comienza con la configuración básica y las instrucciones personales, familiarízate con los comandos slash y variables de chat, experimenta con prompts personalizados, y finalmente explora las posibilidades del MCP cuando tus necesidades lo requieran.

- **Configuración**: Ajusta el comportamiento del auto-completado y del chat.
- **Instrucciones**: Detalla el uso de librerías, lenguajes, documentación, etc.
- **Prompts**: Crea una biblioteca de prompts reutilizables.
- **Modos de chat**: Usa modos de chat personalizados para situaciones o casos concretos.
- **Herramientas**: Instala servidores MCP y usa sus herramientas.

La **inversión de tiempo** se paga rápidamente. Un agente que entiende tu proyecto específico, tus preferencias de código, y tiene acceso a las herramientas que usas diariamente, se convierte **en el compañero de desarrollo** que realmente necesitas: uno que no solo sugiere, sino que *comprende y contribuye* de forma significativa a tu trabajo.

Te sugiero que eches un vistazo a AIDDbot para que te sirva de inspiración. Es un proyecto que usa GitHub Copilot para generar código, documentación, tests, etc. siguiendo la metodología de desarrollo de AI-Driven Development.

> No es mágIA, es Tecnología.
> Alberto Basalo