Evaluación herramientas IA para desarrollo

ÍNDICE DE CONTENIDOS

1. Introducción

En el campo del desarrollo de software, la integración de tecnologías de inteligencia artificial (IA) está ganando cada vez más relevancia. Las herramientas de IA pueden mejorar la eficiencia, la calidad y la productividad de los equipos de desarrollo.

El objetivo de este proyecto es hacer una selección de herramientas de asistentes de código para poder evaluar de manera exhaustiva estas cuatro herramientas de IA para desarrollo de software.

La integración de herramientas de IA puede aportar diversos beneficios clave, entre los que se incluyen:

Aumento de la productividad:

- Automatización de tareas repetitivas, como el llenado de plantillas de código, la implementación de patrones comunes y la generación de código boilerplate.
- Asistencia en la creación de código mediante sugerencias inteligentes y completado de líneas, lo que permite a los desarrolladores centrarse en la lógica principal.
- Capacidad de las herramientas de IA para procesar y analizar grandes volúmenes de código, identificando oportunidades de mejora y optimización.

Mejora de la calidad del código:

- Sugerencias de refactorización y reestructuración del código para hacerlo más legible, mantenible y escalable.
- Detección temprana de posibles problemas, errores y vulnerabilidades, lo que facilita la corrección y el aseguramiento de la calidad.
- Adherencia a mejores prácticas y estándares de codificación, gracias a los conocimientos integrados en las herramientas de IA.

Aceleración del desarrollo:

- Capacidad de las herramientas de IA para generar código de manera más rápida y eficiente, reduciendo el tiempo invertido en tareas manuales.
- Automatización de procesos clave, como la implementación de nuevas funcionalidades, la migración entre versiones y la integración de bibliotecas y frameworks.
- Mejor aprovechamiento del tiempo de los desarrolladores, quienes pueden dedicarse a tareas de mayor valor agregado.

Mejor experiencia para los desarrolladores:

- Liberación de los desarrolladores de tareas repetitivas y tediosas, lo que les permite centrarse en la resolución de problemas complejos y en la innovación.
- Mayor satisfacción y motivación al delegar en las herramientas de IA las tareas más mecánicas y enfocarse en aspectos más creativos y desafiantes.
- Oportunidad de aprender y desarrollar nuevas habilidades al interactuar con las capacidades de las herramientas de IA.

A lo largo de este proceso de evaluación, analizaremos en profundidad el desempeño de las herramientas seleccionadas en una serie de escenarios representativos del desarrollo de software. Los resultados de estas pruebas nos ayudarán a tomar una decisión informada sobre cuál de estas herramientas de IA se adapta mejor a nuestras necesidades y puede aportar un mayor impacto positivo en nuestro flujo de trabajo.

2. Identificación de herramientas iniciales

Para identificar las principales herramientas de IA candidatas a evaluar en profundidad, establecimos los siguientes criterios de selección:

Criterios de selección de herramientas:

- Popularidad y adopción en la industria del desarrollo de software
- Funcionalidades clave orientadas a asistir en el desarrollo.
- Integración con los principales entornos de desarrollo (IDE)
- Soporte para múltiples lenguajes de programación
- Reconocimiento y reputación en la comunidad de desarrolladores
- Disponibilidad de documentación y recursos de apoyo
- Estabilidad, confiabilidad y actualizaciones regulares

El listado inicial de herramientas consiste en una selección preliminar de asistentes de código, que serán evaluadas y refinadas hasta quedarnos con aquellas que analizaremos en profundidad:

1. **Cursor**. Cursor es un editor de código mejorado por inteligencia artificial diseñado para mejorar la eficiencia del desarrollo de software mediante herramientas como la generación de código en tiempo real, detección de errores y autocompletado inteligente.
2. **Codeium**. Codeium es una herramienta impulsada por inteligencia artificial diseñada para optimizar el proceso de desarrollo de software, ofreciendo funcionalidades como autocompletado de código, generación de código, y un chat integrado dentro del entorno de desarrollo (IDE).
3. **Gitlab Duo**. GitLab Duo es un conjunto de herramientas impulsadas por inteligencia artificial (IA) integrado en la plataforma de GitLab. Las capacidades de GitLab Duo van más allá de ser un asistente de programación, ya que incluyen funciones como sugerencia y explicación de código, explicación de vulnerabilidades, generación de documentación, etc.
4. **GitHub Copilot**. GitHub Copilot es una herramienta asistida por inteligencia artificial diseñada para ayudar a los desarrolladores a escribir código de manera más rápida y eficiente. Integra un sistema de sugerencias en tiempo real, un chat para pedir ayuda específica, realizar correcciones o generar código para tareas complejas. También permite crear documentaciones automáticas de cambios en pull requests y gestionar documentación relevante para equipos en entornos de desarrollo.
5. **Replit**: Replit AI es una herramienta de inteligencia artificial integrada en la plataforma Replit que ayuda a los desarrolladores a escribir código más rápido y de manera más eficiente. Ofrece sugerencias en tiempo real para autocompletar código, generar fragmentos de código completos y asistir en la refactorización del mismo. Permite que varios usuarios trabajen simultáneamente en proyectos mientras reciben sugerencias y asistencia de la IA.

6. **Snyk**: Snyk DeepCode AI es una herramienta avanzada de inteligencia artificial que permite identificar y corregir vulnerabilidades de seguridad en el código. Utiliza una combinación de modelos de lenguaje avanzados y algoritmos de IA simbólica, lo que permite sugerir soluciones más precisas y seguras.
7. **SuperMaven**: SuperMaven es una herramienta diseñada para asistir en la creación y gestión de proyectos de software a través de funcionalidades inteligentes que ayudan a escribir código más rápido, identificar errores y aplicar buenas prácticas de desarrollo. Se integra con diversos entornos de desarrollo y plataformas, ofreciendo sugerencias contextuales, autocompletado de código, y ayuda en la identificación de errores.
8. **Tabnine**: Tabnine es una herramienta de asistencia en el desarrollo diseñada para generar código de manera más rápida y eficiente. Su función principal es proporcionar sugerencias de código inteligentes y contextuales a medida que se escribe, lo que puede incluir líneas completas de código o incluso funciones enteras, mejorando la productividad y reduciendo tareas repetitivas.

3. Selección preliminar de herramientas

Una vez identificadas las herramientas iniciales, decidimos hacer una evaluación inicial para verificar cómo se ajusta cada una a nuestros requerimientos.

Los puntos claves a analizar son:

1. **Generación de código**: Analizar la capacidad de la herramienta para generar código de manera eficiente y precisa, alineándose con las mejores prácticas y optimizando el desarrollo. Esto incluye la facilidad con la que puede crear estructuras de código complejas y el grado de personalización que permite en el proceso de generación.

2. **Refactorización y migración de código**: Analizar cómo la herramienta maneja la refactorización de código, asegurando que pueda reorganizar y optimizar el código existente sin introducir errores. Además, es esencial que facilite la migración de versiones antiguas a nuevas, adaptándose a cambios en el lenguaje o la arquitectura sin perder funcionalidad.

3. **Detección de errores y sugerencia de soluciones**: Verificar la eficacia de la herramienta para detectar errores en el código de manera temprana. Esto incluye la identificación de errores lógicos, sintácticos y de rendimiento, así como la capacidad de sugerir soluciones viables para corregir estos problemas de manera rápida y precisa.

4. **Actualización de dependencias**: Evaluar si la herramienta permite gestionar de manera eficiente las dependencias del proyecto, facilitando la actualización automática o manual de librerías y frameworks.

5. **Integración con el IDE**: Comprobar la facilidad de integración de la herramienta con el entorno de desarrollo integrado (IDE), asegurándose de que funcione de manera fluida y no interfiera con el flujo de trabajo del desarrollador. Esto incluye la compatibilidad con las principales plataformas y la capacidad de personalización de la integración según las necesidades del equipo de desarrollo.

Según estos requerimientos planteados, decidimos hacer una comparativa entre las distintas herramientas.

	Generación de código	Refactorización	Detección de errores	Actualización de dependencias	Integración IDE
Cursor	✔️	✔️	✔️	✔️	❌
Codeium	✔️	✔️	✔️	✔️	✔️
Gitlab Duo	✔️	✔️	✔️	✔️	❌
Copilot	✔️	✔️	✔️	✔️	✔️
Replit	✔️	❌	❌	❌	❌
Snyk	✔️	❌	✔️	❌	✔️
Supermaven	✔️	✔️	✔️	✔️	✔️
Tabnine	✔️	✔️	✔️	✔️	✔️

Después de esta comparativa, decidimos eliminar las siguientes herramientas:

1. **Replit**: Replit es una herramienta enfocada más en el desarrollo web en tiempo real y la colaboración, en lugar de funcionalidades de asistencia de IA para el flujo de trabajo de desarrollo. Si bien es una solución interesante, su enfoque principal no se alinea tan bien con los escenarios de prueba definidos.

2. **Snyk**: Snyk DeepCode AI es una herramienta especializada en el análisis de seguridad y la detección de problemas en el código. Si bien es una capacidad importante, su enfoque está más en la identificación de vulnerabilidades que en la asistencia integral a lo largo del proceso de desarrollo.

3. **GitLab Duo**: GitLab Duo ofrece una integración sólida con GitLab, dónde se aloja nuestro código. Sin embargo, sus funcionalidades como asistente de código son inferiores a las otras herramientas seleccionadas.

4. **Codeium**: Si bien Codeium ofrece capacidades de generación y refactorización de código, sus funcionalidades parecen ser más limitadas en comparación con las herramientas seleccionadas, como su siguiente competidor más cercano, Tabnine.

4. Diseño de plan de evaluación

Para evaluar el desempeño de las herramientas de IA seleccionadas (GitHub Copilot, Tabnine, Supermaven y Cursor), someteremos a cada una de ellas a un plan de evaluación que incluye los siguientes escenarios:

1. **Refactorización de una función.** En esta prueba, proporcionaremos a cada herramienta una función existente en código y les solicitaremos un refactor para mejorar su legibilidad, mantenibilidad y adherencia a buenas prácticas. Evaluaremos la calidad del código refactorizado, la claridad de las mejoras implementadas y el cumplimiento de principios de diseño de software.

2. **Generación de código**. En este escenario, pediremos a cada herramienta que genere un nuevo caso de uso a partir de una especificación funcional. Analizaremos la estructura, la lógica y la calidad del código generado, así como su capacidad para cumplir con los requisitos planteados.

3. **Migración a arquitectura hexagonal.** Para esta prueba, presentaremos a las herramientas un controlador existente del monolito y les solicitaremos que lo migren a una arquitectura hexagonal. Evaluaremos si comprenden los principios clave de esta arquitectura y los aplican correctamente en la implementación.

4. **Actualización de dependencias**. En este caso, proporcionaremos a las herramientas un controlador existente y les pediremos que actualicen la versión de una de las dependencias principales. Valoraremos si logran realizar los cambios necesarios en el controlador de manera adecuada.

5. **Detección de errores.** En esta prueba, presentaremos a las herramientas un controlador con bugs conocidos e indicaremos que los identifiquen y corrijan. Evaluaremos su capacidad para detectar y arreglar los problemas existentes en el código.

6. **Generación de tests.** En este paso, pediremos a cada herramienta que genere una suite de tests para una funcionalidad existente y una funcionalidad nueva. Valoraremos la cobertura de los tests generados.

7. **Optimización de código**. En esta prueba, presentaremos a la IA una funcionalidad con problemas de eficiencia. Se valorará la capacidad de identificación de los problemas de eficiencia, su correcta resolución y las métricas de mejora en tiempo.

4a. Refactorización de una función

Objetivo: Mejorar legibilidad y mantenibilidad de código existente
Evaluación: Calidad del código refactorizado y adherencia a buenas prácticas

En esta prueba, le presentaremos el siguiente código que genera el menú que ve el usuario en el front:

Eres un desarrollador senior de Symfony experto en realizar refactors de código adhiriéndose a las buenas prácticas. Necesito que analices la siguiente función con detenimiento y hagas un refactor para que sea más legible y mantenible.

```
Unset
public function getMenu($menu,$ruta = null,$columnas = 3)
{
       //Instancia entity manager para consultas
       $em = $this->getDoctrine()->getManager();

       if ($user = $this->getUser()) {
         //condiciones para usuario logado
       } elseif ($menuOpcionesCustomPrivado) {
         //condiciones custom
       }

       //obtención items de menú
       $menuOpciones = $this->getDoctrine()
              ->getManager()
              ->getRepository(OttLayoutsMenusOpciones::class)

       //...obtención plantilla según menú
       switch ($menu) {
         case 1:
         case 2:
        case 3:
       //... mas tipos de menús

       // renderizado y return del menu generado
       $html = $this->render($template, ...);
       return $html;
}
```

4b. Generación de código

Objetivo: Crear nuevo caso de uso desde una especificación funcional
Evaluación: Estructura, lógica y cumplimiento de requisitos

En esta prueba, le presentaremos a la IA este prompt para que desarrolle un nuevo endpoint API:

Eres un desarrollador senior de Symfony. Necesito que generes el código completo para implementar un endpoint API utilizando las mejores prácticas y patrones recomendados de Symfony 5.4.

Deberás crear un endpoint para gestionar materiales de biblioteca con las siguientes especificaciones:

Endpoint: **GET /api/v.0.2/bib-materials**

Request Body (JSON):

```
Unset
{
  "thematic": string | null,      // Slug de categoría
  "format": string | null,        // Slug de formato
  "search": string | null,        // Término de búsqueda
  "most_seen": boolean,           // Ordenar por más vistos
  "items": number,                // Items por página (default: 9)
  "language": number,             // ID del idioma
  "page": number                  // Número de página (default: 1)
}
```

Response (JSON):

```
Unset
{
  "status": 200,
  "message": "Materials list",
  "data": {
       "items": [
       {
       "id": number,
       "uuid": string,
       "format": string,
       "url": string,
       "name": string,
       "url_image": string,
       "authors": string[]
       }
       ],
       "pagination": {
       "current_page": number,
       "num_pages": number
       }
  }
}
```

Requisitos específicos:

- Usar atributos de PHP 8 donde sea posible
- Implementar interfaces donde sea apropiado
- Usar tipo de retorno strict
- Implementar excepciones personalizadas
- Usar Doctrine ORM para las consultas o SQL nativo donde sea apropiado
- Implementa AbstractController para el controlador

Los requisitos funcionales son:

- Todos los campos del body son opcionales
- Si no se proporciona thematic, devolver materiales de todas las categorías
- Si no se proporciona format, devolver materiales de todos los formatos
- Si no se proporciona search, no aplicar filtro de búsqueda
- Si most_seen es true, ordenar por número de visualizaciones DESC
- Si most_seen es false o no se proporciona, ordenar por id DESC
- El valor por defecto de items es 9
- El valor por defecto de page es 1

4c. Migración a arquitectura hexagonal

Objetivo: Transformar ruta monolito a arquitectura hexagonal
Evaluación: Comprensión y aplicación de principios

En esta prueba, le presentaremos a las herramientas seleccionadas el siguiente prompt para que migre una funcionalidad de monolito siguiendo las buenas prácticas y implementando arquitectura hexagonal:

Actúa como un desarrollador senior de Symfony experto en aplicar principios de arquitectura sostenibles como los principios SOLID, DDD, CQRS y la arquitectura hexagonal. Necesito que migres una funcionalidad de un foro en un LMS que consiste en diferentes rutas que gestionan el CRUD de un foro.

[RUTAS FORO MONOLITO]

Estructura requerida:
Por favor, genera el código completo siguiendo esta estructura DDD y arquitectura hexagonal:

```
Unset
Forum/
├── Application/
│   ├── Create/
│   ├── Update/
│   ├── Delete/
│   ├── Find/
├── Domain/
│   ├── Responses/
│   │   ├── ForumResponse.php
│   ├── Exceptions/
│   │   ├── ForumPostNotFoundException.php
│   ├── Forum.php
│   ├── ValueObjects/
│   │   ├── ForumId.php
│   │   ├── ForumTitle.php
│   │   └── ForumDescription.php
│   ├── Repository/
│   │   └── ForumRepository.php
└── Infrastructure/
        ├── Persistence/
        │   └── Doctrine/
        │       └── Forum.orm.xml
        │   └── DoctrineForumRepository.php
```

Requisitos técnicos:
Por favor, para cada capa de DDD, genera el código siguiendo las siguientes instrucciones:

1. **Dominio**:
    a. Implementar entidades ricas con lógica de negocio encapsulada
    b. Usar Value Objects para todos los atributos que lo requieran
    c. Definir interfaces de repositorio en el dominio
    d. Implementar servicios de dominio para lógica compleja
2. **Aplicación**:
    a. Implementar patrón CQRS separando commands y queries
    b. Usar DTOs para request y response
    c. Implementar manejo de errores a nivel de aplicación
    d. Incluir eventos de dominio para acciones importantes donde sea oportuno
3. **Infraestructura**:
    a. Implementar repositorios usando Doctrine ORM
    b. Implementar consultas con SQL nativo usando PDO donde sea oportuno la performance
    c. Crear API Controllers usando atributos de Symfony 6
    d. Usar el patrón adaptador para servicios externos

Requisitos no funcionales:

1. **Performance**:
    a. Implementar caché donde sea apropiado
    b. Optimizar queries de base de datos
2. **Seguridad**:
    a. Validación robusta de entrada
    b. Sanitización de datos
3. **Mantenibilidad**:
    a. Logging apropiado.
    b. Control de excepciones

4d. Actualización de dependencias

Objetivo: Actualizar versión de dependencias principales
Evaluación: Capacidad de realizar cambios necesarios en el controlador

En esta prueba, mediremos la capacidad de la herramienta para actualizar versiones de dependencias que se utilicen en el proyecto. Dada una dependencia y un controlador donde se use esa librería se pedirá que el asistente de IA actualice la versión de la librería donde haya breaking changes y se medirá su capacidad de análisis de los cambios requeridos para la nueva versión y la efectividad en realizar dichos cambios.

El prompt que le presentaremos a los asistentes será el siguiente:

Actúa como un desarrollador senior de Symfony y conocedor de librerías de pasarelas de pago como stripe/stripe-php. Necesito que para este proyecto actualices la versión de la librería de v11.0.0 a v16.2.0. Actualiza la versión de la librería y haz los cambios necesarios para que todo siga funcionando correctamente siguiendo la especificación nueva.

4e. Detección de errores

Objetivo: Identificar y corregir bugs conocidos
Evaluación: Capacidad de detección y resolución de problemas

En esta prueba, le presentaremos a las herramientas un código que contenga errores y mediremos su capacidad de detección y resolución de bugs:

Este será el prompt que le presentaremos a las herramientas:

Actúa como un desarrollador senior de Symfony experto en analizar y hacer revisiones de código. Necesito que realices una revisión exhaustiva del código proporcionado, repasando todas las llamadas a diferentes funciones que se hacen, comprobando el tipo de datos que se pasa como argumento a cada una de las funciones y detectando cualquier error que pudiese haber en la función o en las funciones a las que llama esta función:

```
Unset
	public function getInscripcionLocaleByUserAndEdicion(?int $idUser, ?int $idEdicion): ?string
	{
	return $this->entityManager->getRepository(OttInscripciones::class)->findOneBy(['usuario' => $idUser, 'edicion' => $idEdicion])
		?->getIdioma()?->getLocale();
	}
```

4f. Generación de tests

Objetivo: Generar tests que cubren los distintos casos de uso de una funcionalidad
Evaluación: Cobertura de los tests generados y el uso de buenas prácticas

En esta prueba, le pediremos a los diferentes asistentes que generen tests para una funcionalidad nueva y una ya existente:

Actúa como un desarrollador senior de Symfony experto en escribir suites de diferentes tipos de tests como los unitarios, de integración y funcionales para cubrir los casos de uso de una funcionalidad. Tienes amplios conocimientos en patrones de testing como Object mother y en la aplicación de principios como test-driven development.

Debes generar tests para la siguiente funcionalidad de añadir/quitar de favoritos un material de biblioteca, para generar los tests ten en cuenta que debes generar los siguientes para las diferentes capas:

- Tests unitarios para el dominio
- Tests de integración para comandos/queries
- Tests funcionales para controllers

Requisitos Técnicos:

Para generar los tests unitarios de dominio debes utilizar las siguientes librerías: PHPUnit, Mockery, Faker. Los tests deben tener la nomenclatura snake case y deben hacer aserciones.

Para generar los tests de integración, debes usar PHPUnit y crear mocks de los repositorios para poder interactuar con ellos y hacer aserciones sin que eso afecte a la infraestructura real.

Para los tests funcionales, debes emplear Behat para simular peticiones a las rutas api, comprobando su robustez y validando los datos de input pasados a los diferentes endpoints.

4g. Optimización de código

Objetivo: Mejorar la calidad de una función aplicando optimizaciones
Evaluación: La calidad de las optimizaciones sugeridas

En esta prueba, le presentaremos a las herramientas una función que tenga problemas de memoria o que haga operaciones redundantes y/o ineficientes:

Actúa como un desarrollador senior de Symfony experto en análisis y optimización de código. Tienes conocimientos avanzados en mejorar el rendimiento de aplicaciones mediante la identificación y resolución de cálculos ineficientes, problemas de memoria y uso excesivo de recursos. Necesito que revises exhaustivamente el código proporcionado, con especial énfasis en:

1. **Detección de Bugs y Errores Lógicos**: Localiza cualquier bug o error lógico que afecte la funcionalidad del programa y propone correcciones claras para cada caso.
2. **Optimización de Eficiencia y Rendimiento**: Identifica cálculos ineficientes, patrones de carga de memoria innecesaria, y cuellos de botella en la ejecución. Evalúa los algoritmos implementados y su complejidad, y sugiere mejoras para reducir la carga de procesamiento y de memoria.
3. **Mejora de Consultas a la Base de Datos**: Si el código incluye consultas a bases de datos, revisa el uso de Doctrine y cualquier consulta SQL, enfocándote en mejorar la eficiencia de las mismas y reduciendo la cantidad de llamadas necesarias.
4. **Gestión de Memoria y Recursos**: Identifica cualquier uso excesivo de memoria, problemas de liberación de recursos, o sobreuso de objetos. Propón soluciones que optimicen la gestión de recursos y mejoren la escalabilidad.
5. **Revisión de Buenas Prácticas en Symfony**: Asegúrate de que se siguen las buenas prácticas en Symfony en términos de arquitectura, uso de servicios, y configuración de caché, y corrige cualquier desviación que pueda impactar la estabilidad o el rendimiento del código.

Para cada problema identificado, proporciona:

- **Explicación detallada** del problema.
- **Soluciones alternativas** y sus ventajas e inconvenientes.
- **Recomendación final** de la mejor solución en función de los objetivos de rendimiento, eficiencia, y escalabilidad del proyecto.

5. Aplicación plan de evaluación

Una vez creado el plan de evaluación el siguiente paso es aplicar todos los pasos definidos a las diferentes herramientas elegidas y evaluar los resultados obtenidos.

5a. Refactorización de una función

En esta prueba, se evalúa la capacidad de las herramientas seleccionadas para realizar una refactorización efectiva de la función elegida. El objetivo es determinar cuál de las herramientas ofrece una solución más completa, organizada y alineada con buenas prácticas de desarrollo.

Criterios evaluados:

1. **Solución funcional:** Verificar si la herramienta genera una solución funcional que respete toda la lógica original.
2. **Modularización:** Analizar si divide el código en métodos claros y reutilizables.
3. **Eficiencia del flujo de lógica:** Asegurar que la herramienta optimice el flujo del código sin comprometer la legibilidad.
4. **Limpieza del código:** Examinar la claridad, legibilidad y adherencia a estándares de calidad del código.
5. **Cobertura:** Confirmar que el refactor abarca todo el código necesario para cumplir con los objetivos originales.

Resultados obtenidos:

1. **Cursor:**

   Puntos fuertes: Ofreció un refactor completo y funcional, con muy buena modularización, ha dividido el código en métodos claros y específicos que abarcan todas las partes del código original.

   Puntos débiles: Un punto débil puede ser la excesiva modularización que puede introducir cierta complejidad, lo cual puede ser confuso en algunos casos.

2. **Copilot**:

   Puntos fuertes: Generó un refactor funcional y modular, cubriendo todos los aspectos principales del código. Los métodos resultantes son más compactos, simplificando el flujo principal.

   Puntos débiles: Algunos métodos combinan diferentes lógicas que compromete un poco la claridad del flujo.

3. **Tabnine**:

   Puntos fuertes: Generó un código más limpio y con un flujo mejor que el original, modularizando correctamente la lógica.

   Puntos débiles: Pese a múltiples intentos, no logró generar un código completamente funcional, dejando métodos sin implementar

4. **Supermaven**:

   Puntos fuertes: Generó un código limpio con métodos bien definidos, introdujo constantes mejorando la mantenibilidad y manejo de excepciones para errores como menús inválidos.

   Puntos débiles: Algunos métodos combinan diferentes lógicas como la de filtros que hace más difícil el seguimiento del flujo completo.

Comparación de herramientas

	Cursor	Copilot	Tabnine	Supermaven
Funcionalidad	**Correcta**	**Correcta**	**Incorrecta**	**Correcta**
Modularización	**Muy buena**	**Buena**	-	**Buena**
Claridad	**Alta**	**Media**	-	**Media**
Buenas prácticas	**Buenas**	**Buenas**	-	**Muy buenas**
Cobertura	**Completa**	**Completa**	**Parcial**	**Completa**

Conclusión:

De las herramientas evaluadas, **Cursor** y **Supermaven** destacan como las más completas en términos de funcionalidad, claridad y adherencia a buenas prácticas.

Supermaven mejora la mantenibilidad del código con el uso de constantes y manejo de excepciones, aplicando buenas prácticas. Sin embargo, **Cursor** se posiciona como la mejor opción al proporcionar un refactor más detallado y preciso, con una modularización superior, claridad del flujo y una cobertura completa de toda la funcionalidad requerida.

En este análisis, **Cursor** es la herramienta ganadora, ya que logra un equilibrio perfecto entre claridad, organización, buenas prácticas y funcionalidad, superando ligeramente a **Supermaven** al optimizar la estructura del código sin comprometer la legibilidad.

5b. Generación de código

En esta prueba, se evalúa la capacidad de las herramientas seleccionadas para generar código funcional a partir de instrucciones o descripciones específicas. El objetivo es determinar cuál herramienta produce un código más preciso, estructurado y alineado con los requerimientos proporcionados.

Criterios evaluados:

1. **Exactitud:** Verificar si el código generado cumple completamente con los requerimientos definidos.
2. **Calidad del diseño:** Analizar la estructura y organización del código generado.
3. **Legibilidad:** Evaluar la claridad y facilidad de comprensión del código.
4. **Adaptabilidad:** Revisar si el código generado es fácilmente ajustable para incorporar nuevos requerimientos o cambios.

Resultados obtenidos:

1. **Cursor**:

   Puntos fuertes: El código generado cumple los requisitos funcionales, implementando DTO para la entrada/salida de datos, validaciones de datos de entrada. Separa la lógica en un servicio y estructura correctamente el código, facilitando la legibilidad del código.

   Puntos débiles: No implementa el patrón repository, por lo que hay una acoplamiento a Doctrine y aunque se manejan errores genéricos, no se implementan excepciones personalizadas.

2. **Copilot**:

   Puntos fuertes: Generó una estructura modular, separando el código en clases separadas (controlador y repositorio). Uso de buenas prácticas de symfony como inyección de dependencias.

   Puntos débiles: Falla en la implementación de los requisitos funcionales como los DTO, validaciones y añade toda la lógica a una función del repositorio.

3. **Tabnine**:

   Puntos fuertes: Generó un código utilizando buenas prácticas como la inyección de dependencias, generación automática de documentación con OpenAPI, estructura el código separando la lógica en un servicio.

Puntos débiles: Falla en la implementación de los requisitos funcionales como los DTO, validaciones, el repositorio contiene código duplicado, lo que dificulta el mantenimiento.

4. **Supermaven**:

   Puntos fuertes: El código implementa los DTO con validaciones, utiliza inyección de dependencias y aplica el patrón repository, creando una interfaz con los métodos del repositorio desacoplados de doctrine.

   Puntos débiles: No estructura correctamente el código, ya que la lógica lo añade directamente en el repositorio, sin crear un servicio, mezclando la lógica de negocio con consultas SQL.

Comparación de herramientas:

	Cursor	Copilot	Tabnine	Supermaven
Exactitud	**Alta**	**Media**	**Media**	**Media**
Calidad de diseño	**Alta**	**Baja**	**Baja**	**Media**
Legibilidad	**Alta**	**Media**	**Media**	**Media**
Adaptabilidad	**Media**	**Baja**	**Baja**	**Baja**

Conclusión:

En esta prueba, **Cursor** destaca por encima del resto de las herramientas ya que es el que cumple más fielmente los requisitos funcionales, la organización destaca por separar correctamente la capa de infraestructura y acceso a datos de la lógica de negocio.

5c. Migración a arquitectura hexagonal

En esta prueba, se evalúa la capacidad de las herramientas seleccionadas para transformar una ruta monolítica en una arquitectura hexagonal. El objetivo es determinar cómo cada herramienta aplica los principios de la arquitectura hexagonal, tales como la separación de responsabilidades, el desacoplamiento entre la lógica de negocio y las dependencias externas, y la flexibilidad para integrar nuevos componentes o adaptadores sin modificar el núcleo del sistema. A través de este análisis, se busca identificar cuál herramienta es más eficaz para migrar a una arquitectura más modular y escalable.

Criterios evaluados:

1. **Comprensión de la Arquitectura Hexagonal:** Evaluar si la herramienta entiende y aplica los principios de la arquitectura hexagonal, como la separación clara entre el dominio y la infraestructura.
2. **Aplicación de Principios:** Analizar cómo la herramienta implementa estos principios en el código, utilizando interfaces para la interacción con los componentes internos y externos.
3. **Desacoplamiento y Flexibilidad:** Evaluar en qué medida el código generado desacopla la lógica de negocio de los detalles de implementación, permitiendo la fácil integración de nuevos adaptadores o componentes.
4. **Facilidad de Integración:** Observar cómo la herramienta gestiona la integración de componentes externos sin afectar el núcleo del sistema, respetando los principios de la arquitectura hexagonal.

Resultados obtenidos:

1. **Cursor**:

   Puntos fuertes: La migración aplica correctamente los principios de la arquitectura hexagonal, principios SOLID y CQRS. Separa el código correctamente entre las capas de DDD. Crea value objects para las validaciones y eventos de dominio para la trazabilidad.

   Puntos débiles: Falta de manejo robusto de excepciones a nivel de aplicación.

2. **Copilot**:

   Puntos fuertes: Aplica correctamente la arquitectura hexagonal, separación clara de responsabilidades y principios SOLID. Aplica CQRS y hace un buen manejo de excepciones y validaciones mediante value objects.

   Puntos débiles: No hay eventos de dominio y falta de manejo robusto de excepciones a nivel de aplicación.

3. **Tabnine**:

   Puntos fuertes: La migración aplica correctamente los principios de la arquitectura hexagonal, principios SOLID y CQRS. Separa el código correctamente entre las capas de DDD. Crea value objects para las validaciones y eventos de dominio para la trazabilidad.

   Puntos débiles: Falta de manejo robusto de excepciones a nivel de aplicación y controlador.

4. **Supermaven**:

   Puntos fuertes: La migración aplica correctamente los principios de la arquitectura hexagonal, principios SOLID y CQRS. Separa el código correctamente entre las capas de DDD. Crea value objects para las validaciones y eventos de dominio para la trazabilidad.

   Puntos débiles: Falta de manejo robusto de excepciones a nivel de aplicación y controlador.

Comparación de herramientas:

	Cursor	**Copilot**	**Tabnine**	**Supermaven**
Comprensión de la Arquitectura Hexagonal	**Alta**	**Alta**	**Alta**	**Alta**
Aplicación de Principios	**Alta**	**Media**	**Alta**	**Alta**
Desacoplamiento y Flexibilidad	**Alta**	**Alta**	**Alta**	**Alta**
Facilidad de Integración	**Alta**	**Media**	**Alta**	**Alta**

Conclusión:

En esta prueba, las tres herramientas **Cursor, Tabnine** y **Supermaven** están a la par ya que aplican correctamente todos los requisitos funcionales y migran correctamente el código a la nueva arquitectura hexagonal. En este caso, **Copilot** también aplica bien la mayoría de los principios pero se deja una pieza muy importante como son los eventos de dominio.

5d. Actualización de dependencias

En esta prueba, se evalúa la capacidad de las herramientas seleccionadas para realizar la actualización de versiones de dependencias principales en un proyecto existente. El objetivo es determinar cómo cada herramienta maneja el proceso de actualizar bibliotecas o componentes del sistema sin introducir fallos o complejidad innecesaria.

Criterios evaluados:

- **Comprensión de la Gestión de Dependencias:** Evaluar si la herramienta facilita la identificación y actualización de dependencias, considerando su impacto en el sistema.
- **Aplicación de Principios de Mantenimiento:** Analizar cómo la herramienta gestiona las actualizaciones, asegurando que el código siga siendo modular y fácil de mantener tras los cambios.
- **Desacoplamiento y Flexibilidad:** Evaluar cómo la herramienta mantiene el desacoplamiento de la lógica de negocio y las bibliotecas externas, permitiendo una actualización de dependencias sin que se vea afectada la estructura del sistema.
- **Facilidad de Integración de Nuevas Versiones:** Observar cómo la herramienta facilita la integración de nuevas versiones de dependencias sin necesidad de modificar el núcleo del sistema, respetando la estabilidad y funcionalidad existentes.

Resultados obtenidos:

1. **Cursor**:

   Puntos fuertes: Es capaz de identificar y realizar los cambios a realizar para adaptarse a la nueva versión, introduce validaciones adicionales y manejo de errores.

   Puntos débiles: No abstrae la funcionalidad a un servicio nuevo, no utiliza el método de StripeClient nuevo recomendado en esta actualización, sigue usando los métodos antiguos.

2. **Copilot**:

   Puntos fuertes: Identifica y realiza los cambios de la nueva versión, implementando correctamente el StripeClient y deshaciéndo los métodos estáticos. Añade validaciones y manejo de excepciones.

   Puntos débiles: No abstrae la funcionalidad a un servicio nuevo, por lo que queda acoplada al controlador y no es escalable

3. **Tabnine**:

   Puntos fuertes: Identifica correctamente los cambios a realizar e implementa el StripeClient correctamente.

   Puntos débiles: No realiza del todo bien la actualización, dejando claves hardcodeadas. No abstrae la funcionalidad a un servicio nuevo, por lo que queda acoplada al controlador y no es escalable

4. **Supermaven**:

   Puntos fuertes: Identifica y detalla cómo actualizar la dependencia. Implementa el StripeClient correctamente.

   Puntos débiles: No abstrae la funcionalidad a un servicio nuevo, por lo que queda acoplada al controlador y no es escalable. No maneja errores específicos de la API.

Comparación de herramientas:

	Cursor	Copilot	Tabnine	Supermaven
Comprensión de la Gestión de Dependencias	Buena	Muy buena	Muy buena	Muy buena
Aplicación de Principios de Mantenimiento	-	Baja	Baja	Baja
Desacoplamiento y Flexibilidad	Baja	Baja	Baja	Baja
Facilidad de Integración de Nuevas Versiones	Baja	Media	Baja	Media

Conclusión:
En esta prueba, **Copilot** y **Supermaven** son los que más destacan y se acercan en la aplicación de los criterios de evaluación. **Copilot** demuestra un equilibrio sólido entre integración de nuevas versiones y manejo de errores, implementando el uso de StripeClient y deshaciéndose de métodos obsoletos. Además, su propuesta de validaciones y excepciones mejora la calidad del código, lo que lo hace más confiable para futuros cambios. **Supermaven** tiene una implementación técnica correcta, pero carece de un manejo específico de errores de la API.

5e. Detección de errores

En esta prueba, evaluaremos la capacidad de las herramientas seleccionadas para identificar, diagnosticar y proponer soluciones a errores en el código base. La detección y corrección de errores es fundamental para garantizar la estabilidad, seguridad y funcionalidad del sistema, especialmente en entornos complejos donde la lógica de negocio interactúa con dependencias externas y sistemas de terceros.

Criterios evaluados:

1. **Identificación de Errores**
   Se analizará si la herramienta es capaz de detectar errores básicos como variables mal definidas, o incompatibilidades de tipos en el código.
2. **Manejo de Excepciones y Errores de Ejecución**
   Se observará si la herramienta puede identificar y sugerir soluciones para excepciones no manejadas o potenciales errores de ejecución, como acceso a índices inexistentes, null pointers, o timeouts en conexiones externas.
3. **Sugerencias de Corrección**
   Se evaluará la calidad y claridad de las recomendaciones proporcionadas por la herramienta para resolver los errores detectados.
4. **Facilidad de Uso**
   Se considerará cómo la herramienta presenta los errores y sus soluciones, priorizando claridad, organización y accesibilidad de la información para el desarrollador.

Resultados obtenidos:

1. **Cursor**:

   Puntos fuertes: Es capaz de hacer un análisis detallado de la funcionalidad y tipos de datos, identifica precisamente las llamadas críticas. Hace propuestas de mejora específicas, incluyendo validación y logging.

   Puntos débiles: La revisión menciona la necesidad de verificar relaciones entre entidades pero no aporta ejemplos específicos de cómo abordarlo.

2. **Copilot**:

   Puntos fuertes: Análisis metódico y estructurado. Validación de tipos de datos y uso correcto del operador null-safe. Recomendaciones de pruebas unitarias.

   Puntos débiles: Falta de manejo de excepciones al acceder a la base de datos. No sugiere validación de parámetros de entrada ni ejemplos concretos de mejoras.

3. **Tabnine**:

   Puntos fuertes: Estructura clara y análisis detallado de cada función. Validaciones adicionales, uso correcto de null-safe. Mejores prácticas, con código mejorado y documentación.

   Puntos débiles: Falta de manejo de excepciones inesperadas y logging. No recomienda pruebas unitarias ni manejo de resultados inesperados.

4. **Supermaven**:

   Puntos fuertes: Firma del método clara. Análisis del flujo de ejecución y uso adecuado de null-safe. Sugerencias de mejora con validación y manejo de excepciones. Consideración del rendimiento.

   Puntos débiles: No sugiere la implementación de logging detallado.

Comparación de herramientas:

	Cursor	Copilot	Tabnine	Supermaven
Identificación de Errores	**Muy buena**	**Mala**	**Muy buena**	**Muy buena**
Manejo de Excepciones y Errores de Ejecución	**Muy buena**	**Mala**	**Mala**	**Buena**
Sugerencias de Corrección	**Muy buena**	**-**	**Muy buena**	**Muy buena**
Facilidad de Uso	**Media**	**-**	**Alta**	**Alta**

Conclusión:
En esta prueba la única herramienta que no ha sido capaz de detectar el error y sugerir una corrección ha sido **Copilot**. Las demás herramientas, han analizado detalladamente la función y detectado y corregido el error inicial.

Entre estas herramientas, destaca **Cursor**, que además de corregir correctamente el error, añade validaciones de errores inesperados y logging. En su respuesta, falta un nivel de detalle de análisis mayor que sí tienen **Tabnine** y **Supermaven**, pero, en general, es el que presenta una mejor solución.

5f. Generación de tests

El objetivo principal de esta evaluación es analizar la capacidad de generar tests automáticas que cubran los diversos casos de uso de una funcionalidad específica dentro del código.

Criterios Evaluables:

1. **Cobertura de Casos de Uso**
   Se evaluará si los tests generados cubren adecuadamente todos los escenarios relevantes
2. **Manejo de Excepciones y Errores**
   Se evaluará la capacidad de los tests para verificar correctamente los manejos de excepciones, condiciones inesperadas o errores de ejecución
3. **Claridad y Organización del Código de Test**
   Se analizará si los tests están bien organizados, son fáciles de entender y si siguen una estructura coherente. Esto incluye la nomenclatura de los tests, la organización de las pruebas en métodos claros y la legibilidad del código.
4. **Asertividad en las Pruebas**
   Se evaluará la calidad y la precisión de las aserciones en los tests. Las pruebas deben contener aserciones claras que validen los resultados esperados y fallen cuando el comportamiento no sea el esperado.
5. **Cumplimiento de Buenas Prácticas de Testing**
   Se analizará el uso de las mejores prácticas en cuanto a la escritura de tests, como la independencia entre tests, pruebas pequeñas y rápidas, y la minimización de efectos secundarios entre diferentes pruebas.
6. **Cobertura de Casos Negativos y Condiciones de Error**
   Se evaluará si los tests incluyen la verificación de condiciones de error esperadas y situaciones inusuales, como entradas inválidas, parámetros nulos, o problemas de acceso a la base de datos.

Resultados obtenidos:

1. **Cursor**:

   Puntos fuertes: Los tests cubren los casos de éxito y de errores correctamente, utilizan librerías mencionadas en los requisitos y cubriendo tests unitarios, de integración y funcionales. Se siguen principios de buenas prácticas y un enfoque claro en la estructura y las aserciones.

   Puntos débiles: Se manejan las excepciones, pero falta cubrir más casos de excepciones o errores lógicos. Menciona configurar el Behat y PHPUnit, pero no proporciona una guía para ello.

2. **Copilot**:

   Puntos fuertes: Uso correcto de las librerías mencionadas, genera los tres tipos de tests pedidos, con tests detallados y estructurados.

   Puntos débiles: Los tests no cubren los casos de errores, en los tests de integración, las aserciones no verifican los efectos esperados, solo el estado. No proporciona configuración para Behat ni PHPUnit.

3. **Tabnine**:

   Puntos fuertes: La estructura de los tests está bien organizada, cubriendo los tres tipos. Utiliza correctamente las librerías mencionadas.

   Puntos débiles: No se cubren los casos de errores en los tests unitarios, los tests de integración y funcionales no hacen las aserciones correctamente.

4. **Supermaven**:

   Puntos fuertes: Los tests creados cubren los tres tipos de test y usan las librerías recomendadas. Además, valida tanto los casos exitosos como los de error, cubriendo escenarios como la falta de parámetros y errores de usuario.

   Puntos débiles: No utiliza el patrón de ObjectMother, pedido en los requerimientos. Falta mejorar el testing de excepciones en los tests de integración.

Comparación de herramientas:

	Cursor	Copilot	Tabnine	Supermaven
Cobertura de Casos de Uso	**Alta**	**Media**	**Alta**	**Alta**
Manejo de Excepciones y Errores	**Buena**	**Mala**	**Regular**	**Regular**
Claridad y Organización del Código de Test	**Muy buena**	**Muy buena**	**Muy buena**	**Muy buena**
Asertividad en las Pruebas	**Muy buena**	**Buena**	**Regular**	**Muy buena**
Cumplimiento de Buenas Prácticas de Testing	**Alta**	**Alta**	**Alta**	**Regular**
Cobertura de Casos Negativos y Condiciones de Error	**Media**	**Baja**	**Baja**	**Alta**

Conclusión:
En esta prueba **Cursor** destaca por encima de las demás herramientas, aplicando los tres tipos de tests con los requerimientos pedidos y una mayor cobertura de los casos de uso. La estructura generada es clara y bien organizada.

Supermaven también cumple con los requisitos correctamente, pero se queda atrás ya que no aplica el patrón de ObjectMother pedido en los requisitos.

5g. Optimización de código

En este caso de prueba, el objetivo es evaluar y validar las mejoras en la optimización de código dentro de un sistema o módulo específico. La optimización de código se refiere a la práctica de modificar un sistema para mejorar su eficiencia en términos de tiempo de ejecución, consumo de recursos (como memoria y CPU), y mantenimiento, sin afectar su funcionalidad.

Criterios evaluados:

1. **Análisis de rendimiento**: Medir el tiempo de ejecución antes y después de la optimización.
2. **Calidad del código**: Asegurar que el código sigue buenas prácticas y es fácilmente mantenible.
3. **Verificación de funcionalidad**: Confirmar que todas las funcionalidades siguen funcionando como se espera después de los cambios.

Resultados obtenidos:

1. **Cursor**:

   Puntos fuertes: Separación de código en un caso de uso, procesamiento de datos en lotes, código mejor optimizado sin utilizar bucles anidados. Carga de traducciones anticipada.

   Puntos débiles: No se implementan todos los métodos, por lo que llegar al funcionamiento final requiere varios pasos. No se ve mejora en la optimización del tiempo de ejecución ni en la reducción del consumo de recursos, dejando el código menos eficiente para escenarios de alta carga.

2. **Copilot**:

   Puntos fuertes: Aplica streaming de datos en la respuesta, por lo que hay una mejora cuando se tratan grandes cantidades de datos. Aplica caché de doctrine y hace refactor de los bucles.

   Puntos débiles: Requiere varios pasos para llegar a un resultado funcional. No aplica mejoras en cuanto a buenas prácticas de separación de conceptos.

3. **Tabnine**:

   Puntos fuertes: Aplica caché de doctrine y carga de traducciones anticipadas.

   Puntos débiles: No es capaz de proveer una solución funcional, aún dando varios pasos.

4. **Supermaven**:

   Puntos fuertes: Hace refactor de la funcionalidad aplicando buenas prácticas. Aplica caché de doctrine y simplifica las consultas en los bucles anidados.

   Puntos débiles: No da una solución funcional, intenta aplicar un streaming de datos pero se hace uso de métodos inexistentes de la librería de exportación utilizada.

	Cursor	Copilot	Tabnine	Supermaven
Análisis de rendimiento	**Media**	**Muy buena**	**Mala**	**Mala**
Calidad del código	**Buena**	**Regular**	**Mala**	**Buena**
Verificación de funcionalidad	**Regular**	**Muy buena**	**-**	**-**

Conclusión:
En esta prueba, las dos únicas herramientas capaces de proveer una solución funcional han sido **Cursor** y **Copilot**. Cursor por su lado ha hecho un mejor trabajo en aplicar buenas prácticas y organización de código. Ha aplicado técnicas de optimización, pero el resultado no se ha visto muy afectado. Copilot en su lado aunque no haya aplicado buenas prácticas como separamiento de conceptos, ha aplicado streaming de datos, por lo que esta optimización se asegura de que no haya fallos por grandes cantidades de datos.

Tabnine y **Supermaven** no han sido capaces de proveer una solución funcional pero Supermaven en el intento sí que ha aplicado buenas prácticas de programación y ha hecho un refactor correcto.

6. Conclusiones finales

Tras evaluar las distintas herramientas en los diferentes casos y tests planteados, vemos que cada herramienta tiene sus propios beneficios y limitaciones:

1. **Cursor**: Esta herramienta sobresale en la capacidad de generar código limpio, estructurado y modular, respetando buenas prácticas de desarrollo. Al ser un IDE completo, tiene más control del proyecto y sugiere mejores soluciones teniendo en cuenta el contexto del proyecto. Además, ofrece distintas funcionalidades como revisión de errores y una generación de código de una forma mejor integrada.

2. **Copilot**: Destaca por su enfoque en la optimización de rendimiento, como el uso de streaming de datos y mejoras en la respuesta a grandes volúmenes de información. También se desempeña bien en la actualización de dependencias, pero carece de precisión en la detección de errores y en la generación de tests, lo que lo hace menos adecuado para tareas que requieren análisis profundo o una alta cobertura de pruebas.

3. **Supermaven**: Es una herramienta sólida que destaca en la implementación de buenas prácticas de programación y provee soluciones siguiendo buenas prácticas. Su capacidad para realizar refactorizaciones claras y seguir patrones establecidos es un punto fuerte. Sin embargo, sufre limitaciones al intentar optimizaciones avanzadas o proporcionar una solución completamente funcional en escenarios complejos, lo que afecta su versatilidad general.

4. **Tabnine** Si bien tiene fortalezas en migraciones arquitectónicas y detección de errores, sufre en la generación de código funcional y la optimización. Ofrece respuestas útiles pero incompletas, lo que requiere intervención manual significativa para completar las tareas. Es adecuada para tareas menos complejas o cuando se necesita soporte puntual más que soluciones completas.

Vamos a concluir siguiendo un sistema de puntuación ponderada para determinar el desempeño general de cada herramienta.

Metodología de puntuación

1. **Asignar puntuación**: A cada prueba se le asigna un peso según su importancia relativa.
2. **Calcular el puntaje ponderado**: Se multiplica la puntuación de cada herramienta en cada prueba por el peso de la prueba y se suman los totales.
3. **Interpretar resultados**: La herramienta con mayor puntaje final será la ganadora.

Pesos por prueba

1. Refactorización de una función: 15%
2. Generación de código: 20%
3. Migración a arquitectura hexagonal: 15%
4. Actualización de dependencias: 10%
5. Detección de errores: 15%
6. Generación de tests: 10%
7. Optimización de código: 15%

Puntuación por herramienta

	Cursor	**Copilot**	**Tabnine**	**Supermaven**
Refactorización	15	12	7	13
Generación de código	18	11	11	14
Migración arquitectura hexagonal	13	10	13	13
Actualización dependencias	5	8	6	8
Detección de errores	13	3	10	11
Generación de tests	9	5	6	7
Optimización	10	13	8	9
Total	**83**	**62**	**61**	**75**