Optimización del Desarrollo de APIs en la Migración de un Monolito: Evaluación del Impacto de la IA

ÍNDICE DE CONTENIDOS

1. Introducción

La evolución tecnológica ha impulsado a muchas organizaciones a migrar de arquitecturas monolíticas hacia sistemas basados en microservicios, con el objetivo de obtener mayor flexibilidad, escalabilidad y eficiencia en el desarrollo. No obstante, este proceso conlleva ciertos desafíos, especialmente en la creación y optimización de APIs que permitan una integración eficiente entre los nuevos servicios.

En este contexto, la inteligencia artificial (IA) ha emergido como un recurso clave para mejorar la velocidad y la calidad del desarrollo de APIs. Herramientas basadas en IA pueden automatizar la generación de código, optimizar pruebas, detectar vulnerabilidades y mejorar la documentación, lo que reduce significativamente los tiempos de desarrollo y minimiza errores.

Este informe analiza cómo la IA puede acelerar el desarrollo de APIs durante la migración de un monolito, evaluando su impacto en la productividad de los equipos de desarrollo y la calidad del software final.

El objetivo de este documento es examinar cómo la inteligencia artificial puede mejorar la velocidad y la eficiencia en el desarrollo de la parte API dentro del proceso de migración de un sistema monolítico hacia una arquitectura modular. Se explorarán herramientas, metodologías y casos de uso que evidencian la aplicabilidad de la IA en este contexto, así como los beneficios y desafíos asociados con su implementación.

2. Optimización del Desarrollo API con Cursor y Composer: Herramientas y Métodos Avanzados

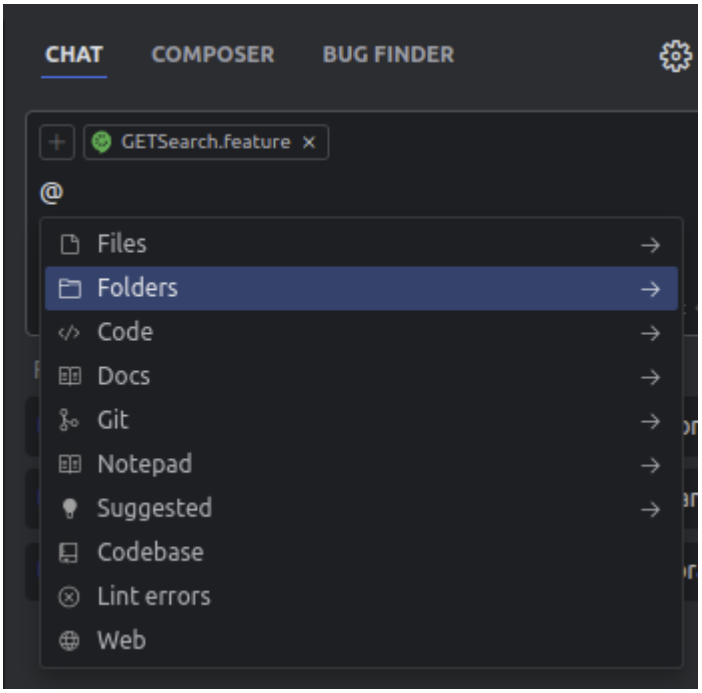
En el desarrollo de la API, estamos utilizando **Cursor** como una herramienta central para agilizar y optimizar el proceso de generación de código. A través de funcionalidades avanzadas como **AutoContext**, **.cursorrules**, **TabTabTab** y el **Modo Agent de Composer**, hemos logrado mejorar tanto la velocidad como la precisión en la creación y corrección del código. A continuación, se explican estas herramientas con más detalle:

2.1 AutoContext (Cursor)

AutoContext en Cursor permite analizar el contexto de las conversaciones y generar automáticamente el código relevante, lo que mejora la eficiencia en el desarrollo de la API. Esto elimina la necesidad de seleccionar manualmente el contexto de código usando los símbolos @. En lugar de eso, AutoContext evalúa la solicitud y ajusta el código generado basándose en el contexto, proporcionando una base más precisa y coherente para cada parte del desarrollo.

Funcionalidad: Utiliza embeddings y modelos personalizados para identificar las áreas relevantes del código y extraer fragmentos útiles de la base de código sin intervención manual.

Beneficio: Mejora la precisión al generar solo el código necesario, adaptado a las conversaciones del flujo de trabajo.



2.2 .cursorrules (Cursor)

El archivo **.cursorrules** es una herramienta que nos permite definir y seguir reglas específicas de estructura de carpetas, nomenclatura de archivos y otras convenciones del proyecto. Estas reglas son esenciales para mantener la coherencia a lo largo del proyecto, asegurando que el código generado cumpla con las normas internas del equipo.

Funcionalidad: Con **.cursorrules**, podemos configurar una estructura predeterminada que guíe la creación del código, como las ubicaciones de las clases, nombres de las funciones y las dependencias necesarias.

Beneficio: La implementación de **.cursorrules** ayuda a que la IA siga una estructura coherente que facilita la integración del código y mejora la calidad general del proyecto.

2.3 Tab Tab Tab (Cursor)

TabTabTab es una función de Cursor que permite autocompletar y corregir automáticamente el código basado en el contexto del proyecto. Cuando se detecta un error o falta de información en el código, se pueden aplicar correcciones al instante con un solo clic.

Funcionalidad: Es utilizado principalmente para corregir errores menores y optimizar fragmentos repetitivos de código. Al reconocer patrones de código defectuosos o incompletos, **TabTabTab** puede ofrecer soluciones inmediatas.

Beneficio: Esto reduce el tiempo que se necesita para corregir errores, aumentando la velocidad de desarrollo al eliminar la necesidad de realizar correcciones manuales frecuentes.

2.4 Modo Agent (Composer)

El **Modo Agent** de Composer es una funcionalidad que permite desglosar el código en pasos pequeños, generando partes del código en lugar de hacerlo de una sola vez. Esto es especialmente útil cuando se trabaja con lógicas complejas que requieren personalización.

Funcionalidad: El **Modo Agent** de Composer se activa para dividir el flujo de trabajo en tareas pequeñas y específicas, lo que permite un desarrollo más ordenado y fácil de manejar.

Beneficio: Esto optimiza el proceso de desarrollo al permitir una mejor gestión de la lógica de negocio, asegurando que cada fragmento de código se desarrolle correctamente antes de pasar al siguiente. Además, mejora la precisión, ya que se corrigen errores en fragmentos más pequeños de manera incremental.

Resumen de la combinación de estas herramientas

Al integrar **AutoContext**, **.cursorrules**, **TabTabTab** y el **Modo Agent**, hemos creado un flujo de trabajo más eficiente y organizado para el desarrollo de la API. Estas herramientas no solo optimizan la generación del código, sino que también mejoran la precisión, la coherencia y la velocidad del proceso de desarrollo.

- **AutoContext** agiliza la adaptación del código al contexto de la conversación.
- **.cursorrules** asegura que el código siga las normas internas del proyecto.
- **TabTabTab** permite correcciones automáticas rápidas y efectivas.
- **Modo Agent** facilita la descomposición de tareas complejas en pasos más manejables.

Estas funcionalidades combinadas han permitido reducir el tiempo de desarrollo, aumentar la calidad del código y disminuir la necesidad de intervención manual, especialmente en tareas repetitivas y estructuradas.

3. Introducción al Proceso de Desarrollo de una API

Durante el desarrollo de una API, dividimos cada endpoint en tres bloques esenciales: **Controlador**, **Caso de uso** y **Test Behat**. Estos bloques permiten una gestión organizada del flujo de desarrollo y nos aseguran que cada parte del código cumpla con los requisitos funcionales y de calidad.

3.1 Controlador

El **Controlador** se encarga de recibir las solicitudes del cliente y gestionar la interacción inicial con el sistema. Su responsabilidad principal es validar y procesar los datos de entrada, pasar la información necesaria a la capa de **Caso de uso** y devolver la respuesta adecuada al cliente. También se ocupa de manejar excepciones y errores, actuando como la capa de orquestación entre la solicitud y la lógica del negocio.

3.2 Caso de uso

En la capa de **Caso de uso**, aplicamos el patrón **CQRS** (Command Query Responsibility Segregation) para separar las operaciones de lectura y escritura. Esta separación nos permite optimizar y gestionar de manera independiente las solicitudes que alteran el estado de la aplicación (commands) y las que simplemente lo leen (queries).

- **Commands**: Encapsulan las solicitudes que realizan cambios en el sistema, como crear, actualizar o eliminar recursos.
- **Queries**: Se encargan de obtener datos sin modificar el estado de la aplicación, optimizando la eficiencia en la lectura.

Además, utilizamos **Response** para estructurar y devolver las respuestas de manera coherente, independientemente del tipo de acción realizada. Las respuestas se gestionan de manera consistente, mejorando la interoperabilidad y facilitando el mantenimiento del sistema.

Otra herramienta clave en esta capa es **Criteria Creator**, que nos permite construir criterios de búsqueda y filtrado de manera flexible y reutilizable. Con **Criteria Creator**, podemos definir consultas complejas de forma declarativa, mejorando la legibilidad y la flexibilidad de las operaciones de lectura.

3.3 Test Behat

Test Behat es nuestra herramienta de pruebas automatizadas de comportamiento. Con **Behat**, escribimos pruebas de aceptación que validan que la API se comporta correctamente desde la perspectiva del usuario final. Estas pruebas no solo aseguran que los endpoints devuelvan las respuestas correctas, sino que también verifican la lógica de negocio implementada en los casos de uso, validando la interacción entre las capas del sistema.

4. Evaluación del Resultado

En este punto, analizamos los resultados obtenidos tras el desarrollo de cuatro endpoints con diferentes niveles de dificultad: baja, media, media-alta y alta. A continuación, se presenta un resumen de los dos primeros endpoints (baja y media) para ilustrar cómo la inteligencia artificial (IA) y las herramientas utilizadas impactaron el proceso de desarrollo.

4.1 Evaluación del Primer Endpoint (Dificultad Baja)

Funcionalidades aplicadas:

- Uso de **Composer** para la generación del código base.
- Sin reglas personalizadas ni herramientas avanzadas de IA.

Resultados obtenidos:

- **Controlador:** 60-70% generado por la IA.
- **CQRS:** 90% generado por la IA.
- **Tests Behat:** 99% generado por la IA.
- **Caso de uso:** 20% generado por la IA, el resto fue corregido manualmente.

Cobertura de código:

- **Alto porcentaje de tareas repetitivas automatizadas.**
- **Problemas en la lógica de negocio del caso de uso**, que requirió intervención manual.

4.2 Evaluación del Segundo Endpoint (Dificultad Media)

Funcionalidades aplicadas:

- Uso de **Cursor** con **.cursorrules** para personalización.
- Optimización del flujo de trabajo con **Notepad** y el **Modo Agent** de Cursor.

Resultados obtenidos:

- **Controlador:** 60-70% generado por la IA.
- **CQRS:** 90% generado por la IA.
- **Tests Behat:** 99% generado por la IA.
- **Caso de uso:** 50% generado por la IA (mejor que en el primer endpoint).

Cobertura de código:

- **Mejora en la generación del caso de uso** gracias a la personalización del flujo de trabajo.
- **Reducción significativa en el tiempo de desarrollo** gracias al uso de herramientas personalizadas.

4.3 Evaluación del Tercer Endpoint (Dificultad Media-Alta)

Funcionalidades aplicadas:

- Uso combinado de **Cursor** y **Composer**.
- **Notepad**, **.cursorrules**, y **Modo Agent** para desglosar tareas complejas en pasos más pequeños.

Resultados obtenidos:

- **Controlador:** 50% generado por la IA.
- **CQRS:** 80% generado por la IA.
- **Tests Behat:** 90% generado por la IA.
- **Caso de uso:** 40% generado por la IA.

Cobertura de código:

- La IA mostró buena eficiencia en la generación de tareas repetitivas, pero la **lógica compleja del caso de uso** (especialmente en la parte de negocio) no fue automatizada correctamente.

4.4 Evaluación del Último Endpoint (Dificultad Alta)

Funcionalidades aplicadas:

- Uso combinado de **Cursor** y **Composer**, con las mismas herramientas del resto de los endpoints: **.cursorrules**, **Notepad**, y **Modo Agent**.
- **Intervención manual intensiva** debido a la alta complejidad de la lógica del caso de uso.

Resultados obtenidos:

- **Controlador:** 30% generado por la IA.
- **CQRS:** 50% generado por la IA.
- **Tests Behat:** 99% generado por la IA.
- **Caso de uso:** No se logró generar correctamente debido a la alta complejidad.

Cobertura de código:

- Aunque la IA fue eficaz para tareas estructurales (controlador, CQRS y tests), **la lógica de negocio del caso de uso fue imposible de automatizar completamente**. La intervención manual fue crucial.

4.5 Resumen General:

- **Funcionalidades aplicadas:**
  En todos los casos, se utilizaron herramientas como **Composer**, **Cursor**, **Notepad**, **Modo Agent** y **.cursorrules** para optimizar el flujo de trabajo y la personalización del código. Las funcionalidades de IA fueron útiles principalmente para tareas repetitivas como la generación de controladores, implementación de CQRS y la creación de tests Behat.

- **Resultados:**
  La IA fue altamente eficiente en la automatización de tareas repetitivas, pero mostró limitaciones claras cuando se trató de lógica de negocio más compleja, especialmente en los casos de uso específicos.

- **Cobertura de código:**
  La IA alcanzó una alta cobertura en tareas repetitivas, pero **la cobertura del caso de uso fue menor** y requirió intervención manual significativa, especialmente en los casos de dificultad media-alta y alta.

5. Resultado de la Evaluación

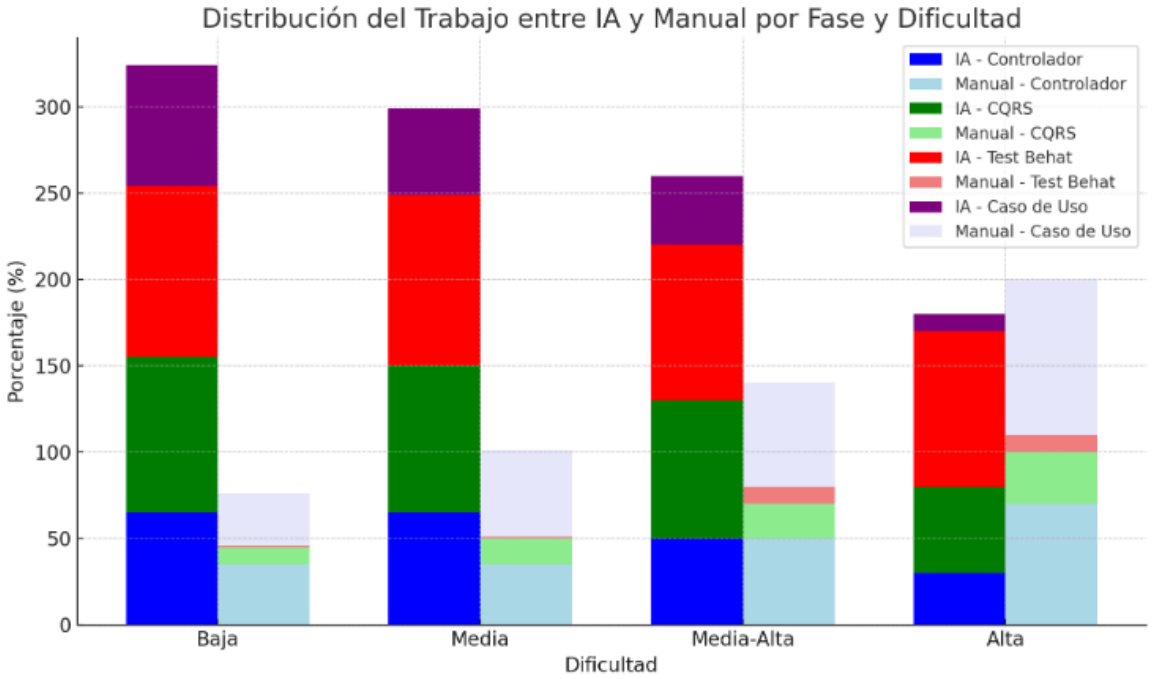
En la siguiente tabla se presentan los resultados obtenidos tras la evaluación del proceso de desarrollo de los cuatro endpoints con diferentes niveles de dificultad. Estos resultados reflejan las métricas clave mencionadas y muestran el impacto de las herramientas de IA en términos de tiempo de desarrollo, cobertura de código, calidad y facilidad de uso para estudiantes.

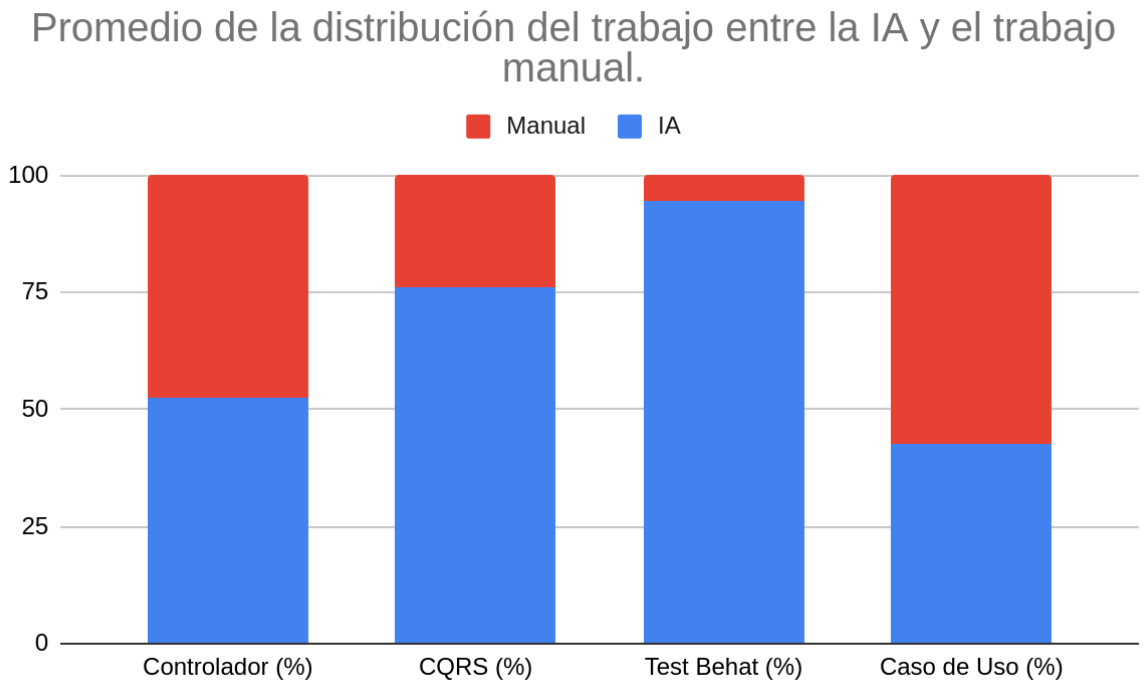
5.1 Tabla de resultados

Dificultad	Controlador (IA %)	Controlador (Manual %)	CQRS (IA %)	CQRS (Manual %)	Test Behat (IA %)	Test Behat (Manual %)	Caso de Uso (IA %)	Caso de Uso (Manual %)
Baja	65	35	90	10	99	1	20(70)	80(30)
Media	65	35	85	15	99	1	50	50
Media-Alta	50	50	80	20	90	10	40	60
Alta	30	70	50	50	90	10	10	90
Medias	**52,5**	**47,5**	**76,25**	**18,75**	**94,5**	**5,5**	**42,5**	**57,5**

#En el primer endpoint, como es la primera prueba, no hemos aplicado muchas funcionalidades, por eso el caso de uso es bajo, pero después de tener más práctica, la cobertura del código puede llegar hasta un 70%.

5.2 Gráficos


Distribución del Trabajo entre IA y Manual por Fase y Dificultad
IA - Controlador
Manual - Controlador
IA - CQRS
Manual - CQRS
IA - Test Behat
Manual - Test Behat
IA - Caso de Uso
Manual - Caso de Uso
Porcentaje (%)
0
50
100
150
200
250
300
Baja
Media
Media-Alta
Alta
Dificultad


Promedio de la distribución del trabajo entre la IA y el trabajo manual.
Manual
IA
100
75
50
25
0
Controlador (%)
CQRS (%)
Test Behat (%)
Caso de Uso (%)

Conclusión General:

La automatización de tareas repetitivas y la estructura básica del desarrollo de APIs mediante herramientas de IA ha demostrado ser efectiva y eficiente, acelerando el proceso de desarrollo en varios casos. Sin embargo, en tareas más complejas, especialmente en la implementación de casos de uso con lógica de negocio especializada, la intervención manual sigue siendo esencial. La combinación de IA para tareas estructurales y trabajo manual para las fases complejas parece ser la mejor estrategia para garantizar la calidad del código y la eficiencia del proceso.

6. Conclusión

Este informe ha evaluado el impacto de la inteligencia artificial (IA) en el desarrollo de APIs durante la migración de un monolito a una arquitectura modular. A través de la implementación de herramientas como **Cursor**, **Composer**, **AutoContext**, **.cursorrules**, **TabTabTab** y el **Modo Agent**, se ha logrado optimizar el proceso de desarrollo de APIs.

En las tareas más simples y repetitivas, la IA ha acelerado significativamente el desarrollo. En los endpoints de baja y media complejidad, la IA generó hasta un 99% del código, reduciendo el tiempo de desarrollo considerablemente. Sin embargo, en los casos más complejos, como la lógica de negocio avanzada, la IA aún requiere intervención manual para garantizar la calidad del código.

Aunque la IA puede automatizar gran parte del proceso en fases iniciales, como la creación de código base y pruebas, las tareas más complejas siguen necesitando supervisión y ajustes manuales. En este sentido, una estrategia combinada de IA y trabajo manual sigue siendo la más eficiente.

En cuanto a la automatización de endpoints, es posible automatizar la mayoría de ellos, pero se necesita un enfoque gradual y supervisado. Los estudiantes, con una comprensión básica del desarrollo de APIs, pueden beneficiarse de estas herramientas, pero todavía deben tener supervisión para manejar la lógica de negocio más compleja.

Para lograr una automatización eficiente, se sugiere el siguiente enfoque dividido en fases:

- **Fase 1**: Automatización del código base, como controladores, implementación de CQRS y pruebas.
- **Fase 2**: Uso de IA para identificar y corregir errores en el código generado.
- **Fase 3**: Supervisión manual y corrección de partes del código que involucren lógica compleja.

En resumen, la IA mejora significativamente la productividad, especialmente en tareas repetitivas, pero aún no puede reemplazar completamente la intervención humana en casos más complejos. Una combinación de herramientas de IA y supervisión manual es la clave para optimizar el desarrollo de APIs, incluso para estudiantes.

7. Proceso de Automatización

Para lograr un flujo de trabajo automatizado eficiente en el desarrollo de APIs utilizando IA, se sigue una serie de pasos estructurados que ayudan a optimizar la productividad, mientras se mantienen las revisiones manuales donde son necesarias. A continuación se detalla el proceso de automatización basado en la evaluación realizada:

1. **Guardar la función en @Notepad y agregar en el contexto**:
   Se inicia guardando cualquier función clave en @Notepad para tener un registro claro y accesible de las funcionalidades que se van a utilizar y revisar durante el proceso de desarrollo.

2. **Generación del controlador del API con la IA**:
   Una vez guardada la función en @Notepad, se solicita a la IA que genere el controlador del API. Para ello, se proporciona el endpoint y ejemplos de otros controladores utilizando la estructura organizada en @Folders. La IA se encargará de crear un controlador que siga la misma lógica y formato que los ejemplos proporcionados.

3. **Revisión del controlador**:
   Después de que la IA genere el controlador, se procede a revisar si maneja adecuadamente las excepciones y otros detalles del flujo. Es importante verificar que el controlador esté correctamente estructurado y sea robusto para posibles fallos, adaptándolo a las necesidades del proyecto.

4. **Generación del caso de uso, CQRS, Response, Repository y Criteria Creator**:
   A continuación, se solicita a la IA que genere el caso de uso, CQRS, Response, Repository y Criteria Creator de la entidad correspondiente. Se pasa como ejemplo la carpeta de la API utilizando @Folders para asegurar que la IA siga la estructura correcta y genere el código de acuerdo con el contexto del proyecto.

5. **Revisión del código generado**:
   A veces, la IA puede no controlar correctamente los nombres de las variables o generar detalles que no se ajustan completamente al contexto. Es necesario revisar y modificar el código, pasando las entidades correspondientes con @File, y ajustando manualmente las variables y funciones para que se adapten correctamente a las funciones guardadas previamente en @Notepad.

6. **Prueba del API**:
   Tras realizar las modificaciones necesarias en el código, se debe probar el API para verificar que funcione correctamente. Si el API no responde como se espera, se deben hacer los ajustes pertinentes antes de continuar.

7. **Generación de tests Behat con la IA**:
   Una vez el API ha sido probado, se solicita a la IA que genere los tests Behat correspondientes. Para esto, se proporcionan ejemplos de pruebas previas utilizando @Folders, permitiendo que la IA genere las pruebas adecuadas para asegurar que todas las funcionalidades del API estén correctamente cubiertas.

8. **Ejecución de los tests**:
   Después de generar los tests, se ejecutan para verificar que todos pasen correctamente. Cualquier test que falle debe ser revisado y corregido, ajustando el código según sea necesario.

9. **Composer fix y subida del código**:
   Finalmente, se ejecuta el comando `composer fix` para asegurar que el código esté bien formateado y siga los estándares de calidad. Una vez realizada esta validación, el código puede ser subido al repositorio, completando el proceso de desarrollo automatizado.

Este proceso de automatización, que combina el uso de la inteligencia artificial con la intervención manual, permite optimizar el desarrollo de APIs de manera eficiente, garantizando que se cumplan tanto las necesidades del proyecto como los estándares de calidad. La clave es encontrar el equilibrio entre la automatización y la personalización manual, asegurando que las complejidades del proyecto sean abordadas correctamente sin perder la eficiencia.