UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Un análisis crítico sobre el uso de herramientas generativas de código en los procesos de desarrollo de Software

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Camps Edo, Alejandro

Tutor/a: Ramírez Quintana, María José

CURSO ACADÉMICO: 2023/2024

Resumen

Las herramientas de generación automática de código, de igual manera a todos los campos que emplean Inteligencia Artificial, avanza a un ritmo frenético. Es nuestro deber como personas ligadas al mundo de la tecnología conocer y masterizar el uso de estas herramientas, que con total seguridad serán un elemento clave en el futuro del desarrollo del software. En este trabajo no solo conoceremos más acerca de ellas y de toda la tecnología que les subyace, también veremos cuál es la mejor manera de emplearlas, las más conocidas, sus características y haremos un repaso a algunos estudios para saber en qué punto se encuentran estas herramientas de cara a su integración completa en los procesos de desarrollo de software, acercando más a todo el mundo a estas nuevas tecnologías. Adicionalmente, llevaremos a cabo un pequeño estudio práctico utilizando las herramientas disponibles a nuestro alcance, aplicando las técnicas y métodos que hemos aprendido, para comprobar de primera mano cómo se implementan y cómo se comportan en situaciones reales. Esto nos permitirá conocerlas más de cerca, evaluar su eficacia y comprender mejor sus fortalezas y limitaciones.

Resum

Les eines de generació automàtica de codi, de la mateixa manera que tots els camps que empren Intel·ligència Artificial, avancen a un ritme frenètic. És el nostre deure, com a persones lligades al món de la tecnologia, conéixer i dominar l'ús d'estes eines, que amb total seguretat seran un element clau en el futur del desenvolupament del programari. En aquest treball, no sols coneixerem més sobre elles i sobre tota la tecnologia que els sustenta, sinó que també veurem quina és la millor manera d'emprar-les, les més conegudes, les seues característiques i farem una revisió d'alguns estudis per a saber en quin punt es troben aquestes eines de cara a la seua integració completa en els processos de desenvolupament de programari, apropant aquestes noves tecnologies a tothom. A més, portarem a terme un xicotet estudi pràctic utilitzant les eines disponibles al nostre abast, aplicant les tècniques i mètodes que hem aprés, per a comprovar de primera mà com s'implementen i com es comporten en situacions reals. Això ens permetrà conéixer-les de més a prop, avaluar la seua eficàcia i comprendre millor les seues fortaleses i limitacions.

Abstract

Automatic code generation tools, like all fields that employ Artificial Intelligence, are advancing at a frenetic pace. It is our duty, as people connected to the world of technology, to know and master the use of these tools, which will undoubtedly be a key element in the future of software development. In this work, we will not only learn more about them and the technology that underlies them, but we will also see the best ways to use them, identify the most well-known tools, their characteristics, and review some studies to determine where these tools stand in terms of their full integration into software development processes, bringing these new technologies closer to everyone. Additionally, we will conduct a small practical study using the tools available to us, applying the techniques and methods we have learned to see firsthand how they are implemented and how they behave in real situations. This will allow us to get to know them more closely, evaluate their effectiveness, and better understand their strengths and limitations.

Índice del trabajo

1. Introducción

El área del desarrollo del software es un mundo increíblemente complejo y dinámico, caracterizado por un continuo desarrollo de todos los aspectos, herramientas, metodologías y técnicas que lo componen. Esta evolución constante no solo es un signo de progreso, sino una necesidad imperativa para mantenerse relevante y competitivo en una industria que avanza a pasos agigantados. Las mejoras en el desarrollo de software son de carácter vital, especialmente para aquellos especializados en el campo, ya que la capacidad de adaptarse a nuevas tecnologías y prácticas puede determinar el éxito o fracaso de los proyectos. No solamente los desarrolladores o ingenieros del Software individualmente tienen que conocer la vanguardia en técnicas de programación, entornos de desarrollo, lenguajes o técnicas de trabajo en equipo, sino que las empresas deben tener integradas en tus equipos todas estas técnicas y asegurarte que funcionen correctamente, pudiendo así ofrecer las mejores prestaciones a la hora de manejar los distintos proyectos que posean.

Paralelamente, en el mundo de la tecnología, estrechamente unido al del desarrollo del Software, la clara vanguardia estos últimos años es la Inteligencia Artificial (IA), la cual se ha extendido rápidamente a todos los campos que este mundo compone. No hay prácticamente ningún ámbito en el cual no se haya construido alguna IA para que haga cualquier cosa en relación con éste. Lo que puede dar a entender una falsa (o no tan falsa) sensación de que, si no existe una IA implicada en un campo, este no está tan desarrollado como debería.

Los Modelos Grandes de Lenguaje o *Large Lenguaje Models* (LLM), como GPT-3 y sus sucesores, representan un hito en el procesamiento del lenguaje natural (*Natural Language Processing*, NLP). Utilizando arquitecturas avanzadas basadas en transformadores, estos modelos pueden predecir la siguiente palabra en una secuencia, generando así textos que varían desde simples correos electrónicos hasta complejas líneas de código. Su capacidad para aprender de vastos conjuntos de datos y adaptar su conocimiento a contextos específicos, los convierte en herramientas muy poderosas para una amplia variedad de aplicaciones y automatización de tareas.

Cuando hablamos de LLM no solo estamos hablando del mundo de la tecnología, sino que, en los últimos años, los LLM han revolucionado múltiples campos no solo tecnológicos, sino también de la vida cotidiana mediante la aplicación de técnicas avanzadas de procesamiento del lenguaje natural, que se han extrapolado a prácticamente cualquier tarea que realizamos, desde la más simple a la más compleja. Estos modelos han demostrado una capacidad impresionante para comprender y generar texto en lenguaje natural, siendo capaces de realizar un abanico muy amplio de tareas distintas, desde los modelos conversacionales más básicos, pasando por la generación de imágenes o vídeos, hasta analizar ecografías o radiografías para detectar anomalías de manera automática.

La capacidad de los LLM para generar código ha abierto nuevas posibilidades en la automatización del desarrollo de software, facilitando tareas como la creación de scripts, la corrección de errores y la implementación de algoritmos complejos. Estos modelos no solo permiten ahorrar tiempo y recursos, sino que también pueden ofrecen a los desarrolladores una forma efectiva de aprender y comprender código o lenguajes de programación con los que no están tan familiarizados. Además, los LLM pueden servir

como asistentes inteligentes, sugiriendo mejoras en el código y ayudando a los desarrolladores a evitar errores comunes.

Juntando estas dos visiones, es obvio que la implicación de la IA en el desarrollo del software no solo es posible, sino que es necesaria. Sin embargo, una pregunta fundamental surge en este contexto: ¿qué tan viable es actualmente la implementación de estas tecnologías? Hace solo unos años, la idea de que la IA pudiera generar código funcional y útil parecía una visión lejana y poco realista. Hoy en día, los avances han sido sorprendentes, con herramientas como Copilot de GitHub[1] y Codex de OpenAI[2], que demuestran capacidades impresionantes en la generación de código, asistencia en la depuración y automatización de tareas repetitivas.

A pesar de estos avances, es importante reconocer que aún existe un camino significativo por recorrer. Aunque los LLM han mostrado un desempeño notable, su integración en el desarrollo de software enfrenta varios desafíos. Por un lado, la precisión y la coherencia del código generado aún no alcanzan un nivel que permita confiar plenamente en estos modelos sin supervisión humana. Los desarrolladores deben revisar y, en muchos casos, ajustar el código propuesto para asegurar que cumpla con los requisitos específicos y que esté libre de errores críticos. Los errores producidos por LLM que no están bien ajustados o entrenados son frecuentes, y pueden suceder en cualquier tarea, cómo se irá ilustrando a lo largo del trabajo. Es importante recalcar que, como cualquier tecnología, la producción de errores no es algo único de los LLM, y que durante la historia de la informática los fallos o errores no han sido pocos o simples, lo que nos tiene que recordar, a la hora de ver a LLM hacer cosas raras, que todo empezó así y que hay que dar tiempo a que se desarrollen versiones mejores.

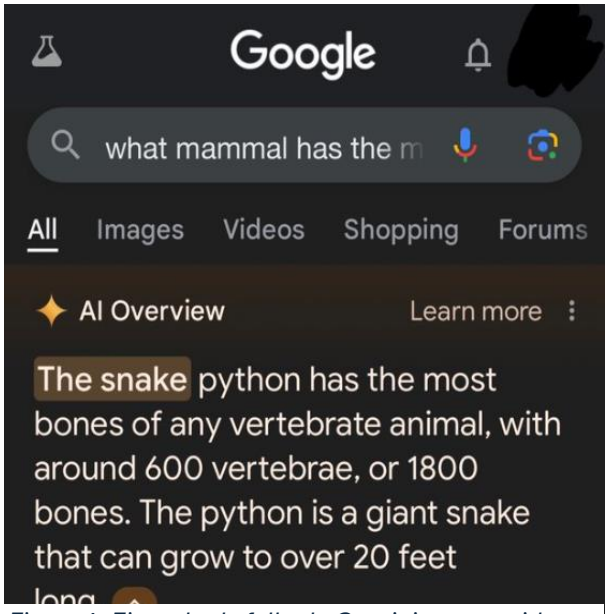


Figura 1. Ejemplo de fallo de Gemini compartido por @RavenTenebris13

Un ejemplo muy reciente es el protagonizado por AI Overview, una función que introdujeron a Gemini[3], la IA desarrollada por Google, destinada a apoyar a los usuarios en las consultas que hacían en el buscador. Pese a que Google avisó de que esta opción estaba en periodo de pruebas, muchos usuarios expusieron sus quejas acerca de que los resultados obtenidos rozaban lo absurdo, como nos muestra un usuario de "X" en la Figura 1, donde esta Inteligencia Artificial asume, de manera errónea, que las serpientes pertenecen al grupo de los mamíferos.

[1] https://github.com/features/copilot
[2] https://openai.com/index/openai-codex/
[3] https://gemini.google.com

Pese a que rápidamente Google mejoró la herramienta y solucionó problemas similares que había estado experimentando, debidos a algunos fallos que sufría AI Overview cuando las preguntas que le hacían no eran muy comunes o concretas, podemos ver como el hecho de "apresurarse", pudo haber dado a la multinacional un pequeño susto.

Además, la viabilidad de estas tecnologías también está condicionada por aspectos éticos y prácticos. La privacidad de los datos utilizados para entrenar estos modelos, la propiedad intelectual del código generado y la transparencia en el proceso de toma de decisiones de la IA son cuestiones que requieren atención cuidadosa. Asimismo, existe una necesidad constante de actualizar y entrenar los modelos con datos actuales y relevantes para mantener su efectividad y adaptabilidad.

Todo esto nos lleva a pensar que antes de incluir la IA en algún campo o tarea, deberíamos, como mínimo, estar seguros de su correcto funcionamiento, excluyendo así posibilidades de fallos notorios como el que acabamos de ver, pero siempre teniendo en cuenta que, como cualquier tecnología, fallos menores pueden suceder cuando hablamos de IA. Nuestro objetivo con este trabajo es arrojar un poco de información sobre cómo estas IAs pueden trabajar en los diferentes estadios del proceso del desarrollo del Software, siendo el principal toda la fase de generación y mantenimiento, ya que la generación de código es la tarea más notoria en la que puede servir de apoyo (también es la que más trabajo supone), pero explorando también posibilidades en otras fases del desarrollo de un producto Software.

La aplicación de LLMs en el desarrollo de software tampoco está libre de errores. De hecho, estudios de carácter similar a este trabajo, ponen en evidencia este aspecto. Como se indica en [1], "The experimentation shows that the three evaluated LLM engines are able to solve the three exams but with the constant supervision of software experts in performing these tasks. Currently, LLM engines need human-expert support to produce running code that is of good quality", donde el autor recalca lo poderosas que pueden ser estas herramientas, pero que, actualmente, necesitan de un humano experto en software para que el producto final sea sólido y fiable.

La capacidad de los LLM para generar código no es algo nuevo, ya que, como se verá más adelante, generan código de la misma manera que lo hacen con el lenguaje natural, por lo que esta capacidad nació prácticamente al mismo tiempo que su capacidad para confeccionar textos, pero obviamente de una forma muy primitiva e imperfecta. Nuestro objetivo con este trabajo es estudiar cómo han evolucionado estas herramientas en el ámbito de la generación de código y comprobar si sus capacidades actuales son óptimas para ofrecer un apoyo fiable.

La adopción de LLM en el desarrollo de software también depende de la aceptación y la comprensión por parte de los desarrolladores y las organizaciones. Muchos usuarios pueden no estar completamente informados sobre las capacidades actuales y las limitaciones de estas tecnologías, como se ha dicho anteriormente, es responsabilidad de los programadores estar actualizados en la vanguardia tecnológica que afecta a las técnicas y herramientas de programación.

Por ello es esencial fomentar una educación continua y proporcionar recursos que ayuden a los desarrolladores a entender y utilizar estas herramientas de manera efectiva y ética, aunque no solo de cara a los programadores, sino que instruir a la sociedad para que

se adapte y entienda el gran avance que suponen los LLM para muchos aspectos y tareas, de igual manera que en su momento se hizo con la informática.

No solamente son interesantes estas herramientas por el hecho de que como Ingenieros de Software o desarrolladores podamos emplearlas como apoyo, sino también como profesores, siendo la capacidad de estas para explicar tanto código en sí como técnicas de programación considerablemente alta. El hecho de que los desarrolladores e ingenieros de Software requieran estar en continuo aprendizaje, ya que como se ha apuntuado anteriormente, es un campo que está en continua evolución, nos empuja a necesitar medios fiables por los cuales aprender, por lo que sus aplicaciones en el ámbito de desarrollo de software son mayores que un simple compañero con el que hacer Pair Programming. Sin embargo, en este trabajo nos centraremos en la parte de generación de código, por lo que, pese a que algunas de las herramientas de manera intrínseca acompañan al código generado con una breve explicación, no se tendrá en cuenta para nuestros fines analíticos.

Es importante destacar que, como todo en este sector, estas herramientas están en constante crecimiento y mejora, por lo que las versiones de las herramientas que se usarán para realizar el estudio en este trabajo pueden mejorar en un corto periodo de tiempo, se indicará siempre que se use o se hable de una herramienta la versión de esta. Siendo conscientes de este hecho, viendo la creciente tendencia del uso de una IA para cualquier tarea, nos parece oportuno abordar este tema con las capacidades actuales, arrojando una visión sobre si se están sobreestimando. Aunque con total certeza los defectos que estas presentan serán solucionados en un futuro, posiblemente cercano.

Centrándonos en este trabajo, nuestro principal objetivo es estudiar los modelos generativos de código, aprendiendo que eran más internamente y observando cómo desempeñan su función, relacionándolo con la ingeniería del Software. No somos los únicos, y como iremos viendo a lo largo del trabajo, muchos estudios se han llevado a cabo con la finalidad de poner a prueba estos modelos, con la diferencia de que este trabajo se centrará en herramientas que han sido entrenadas específicamente para generar código, como veremos más adelante, y en estudiar las herramientas así, conociendo más a fondo el interesante y para muchos, desconocido, mundo de los Modelos de Lenguaje.

Pese a que, el uso de chatbots de carácter general (siendo GPT-3 o GPT-4 como los más conocidos) para generar código está normalizado en la mayoría de los entornos de desarrollo, también se conoce que éstos cometen bastantes errores y no es recomendable utilizar el código generado sin antes realizar una depuración y un testeo. Por esto pensamos que nuestra elección no solo es interesante, pues son herramientas o muy conocidas, o desarrolladas por las empresas más grandes del sector, sino que también es novedosa, tratando solo herramientas que, como se ha indicado anteriormente, están pre-entrenadas exclusivamente para generar código.

También nos centraremos en buscar y analizar herramientas generadoras cuyo uso pueda cómodamente adaptarse a todas las fases del proceso del desarrollo de Software, demostrando que, no solamente en las fases de desarrollo son útiles, sino que su integración en el desarrollo del Software puede ser total. Acercando más el objetivo del trabajo a la propuesta didáctica del grado, más en específico de la especialización escogida, Ingeniería del Software, donde se trabajan todos los pasos y estados del proceso de generación y mantenimiento de un producto Software.

Presentamos también como uno de los capítulos clave los distintos grupos de generadores de código, habiendo dos grandes familias, los modelos conversacionales, a los que les preguntas un fragmento de código amplio y te lo proporcionan, y los asistentes, que mientras escribes el código en tu entorno, te va sugiriendo código para continuar, de igual manera que lo podría hacer un autocompletado de palabras, instalados en prácticamente todos los teclados de dispositivos móviles. Estas dos familias, pese a que se basan en lo mismo, LLM entrenados para generar código, su funcionamiento es completamente diferente, pero ya lo veremos detalladamente más adelante.

Hemos querido centrarnos en Inteligencias Artificiales que hayan sido entrenadas para generación de código, comparando tanto los resultados como los entrenamientos de estas, teniendo en cuenta esta visión y añadiendo una separación entre las herramientas, agrupándolos en dos grandes grupos, los asistentes y los generadores, brindamos un punto de vista nuevo.

Finalmente, en el último capítulo revisamos otros trabajos y/o estudios que también se han centrado en probar la IA en el ámbito de generación de código, siendo estos un punto de apoyo para entender el enfoque dado en este trabajo, veremos diferentes trabajos realizados con diferentes técnicas y distintas herramientas, haciéndonos una idea de la gran variedad de estudios que se pueden realizar y estudiando los distintos resultados que estos logran.

2. Los LLM

Las herramientas que someteremos a análisis en este trabajo son en realidad Grandes Modelos de Lenguaje. En general, los LLMs están siendo utilizados en una amplia variedad de aplicaciones de Inteligencia Artificial. Por ejemplo, plataformas como Jasper[4] utilizan LLM para generar textos para blogs, artículos y contenido de marketing; empresas como Brandwatch[5] y Sprinklr[6] utilizan LLM para analizar y categorizar comentarios y publicaciones en redes sociales para medir el sentimiento del público sobre temas específicos; y herramientas como Grammarly y Turnitin[7] usan LLM para ofrecer sugerencias de mejora en la escritura y detección de plagio.

En específico, vamos a centrarnos en los modelos generativos, que son algoritmos de aprendizaje automático que aprenden la distribución de los datos para generar nuevos datos similares a los originales, con aplicaciones que abarcan desde la creación de contenido hasta la mejora de datos y simulaciones. A diferencia de los modelos discriminativos, que predicen etiquetas a partir de datos, los modelos generativos se centran en replicar la estructura subyacente de los datos.

Como nos cuenta Amazon en su web oficial: "La inteligencia artificial generativa (IA generativa) es un tipo de IA que puede crear ideas y contenidos nuevos, como conversaciones, historias, imágenes, videos y música. Las tecnologías de IA intentan imitar la inteligencia humana en tareas de computación no tradicionales, como el reconocimiento de imágenes, el procesamiento de lenguaje natural y la traducción. La IA generativa es el siguiente paso en la inteligencia artificial."[8]

2.1. Orígenes de los LLM

La historia y origen de los LLM están estrechamente relacionados con el desarrollo de técnicas de NLP y aprendizaje profundo en la inteligencia artificial.

El concepto más básico y fundamental en los LLM es el del lenguaje natural, que es simplemente el lenguaje escrito o hablado que usamos los seres humanos para comunicarnos. El procesamiento y la generación del lenguaje natural son campos de las ciencias computacionales e inteligencia artificial, cuyo objetivo es brindar a las computadoras las herramientas necesarias para comprender, procesar y generar el lenguaje humano (lenguaje natural). Esta combinación de procesamiento y generación del lenguaje natural permite desarrollar un amplio rango de aplicaciones como la traducción de voz y texto, el reconocimiento de voz, el análisis de texto y los chatbots entre otros. Los LLM son modelos de IA que han sido entrenados con grandes cantidades de datos de texto para entender y generar lenguaje humano de manera similar a como lo haría una persona. Por lo tanto, estos modelos están diseñados para realizar una variedad de tareas de NLP, como traducción de idiomas, resumen de textos, respuesta a preguntas y conversaciones.

[4] https://www.jasper.ai/
[5] https://www.brandwatch.com/
[6] https://www.sprinklr.com/
[7] https://www.turnitin.com/
[8] https://aws.amazon.com/es/what-is/generative-ai/

2.2. Historia de los LLM

La historia de la IA y del NLP se remonta a los primeros días de la informática. En 1950, el matemático británico Alan Turing propuso en su ensayo “Computing Machinery and Intelligence” [2] lo que ahora conocemos como el Test de Turing. Este experimento, diseñado para evaluar la capacidad de una máquina para exhibir un comportamiento inteligente, y basado en aislar a un humano con una computadora para que este determine si piensa que está tratando con una máquina o con otro ser humano, sentó las bases para el desarrollo de la Inteligencia Artificial.

Desde 1960 a 1980

En 1966, se creó el primer chatbot de la historia, Eliza [3], creado por Joseph Weizenbaum en el Instituto de Tecnología de Massachusetts. Eliza se basaba en patrones de sustitución para generar respuestas, lo que le permitía imitar el lenguaje natural. Aunque su capacidad de procesamiento y su repertorio para hacer frente a distintas casuísticas planteadas por los usuarios eran limitados, su enfoque simulando conversaciones humanas fue realmente innovador para la época, y sentó las bases para los chatbots como los conocemos hoy en día.

En los años 60 y 70 se desarrollaron los primeros algoritmos para la traducción automática y el análisis sintáctico. Estos sistemas se basaban en reglas explícitas escritas a mano y carecían de la capacidad de aprender de grandes volúmenes de datos.

Década de los 90

A partir de los años 90, la disponibilidad de grandes cantidades de datos digitales y el aumento del poder de cómputo permitieron el uso de métodos estadísticos. Modelos como el Modelo de Máxima Entropía y los Modelos Ocultos de Markov (HMM) comenzaron a utilizarse para tareas como el etiquetado de partes del discurso y el reconocimiento de voz. Estos modelos eran más flexibles que los sistemas basados en reglas, pero aún tenían limitaciones en cuanto a la complejidad del lenguaje que podían manejar.

Desde 2000 hasta la actualidad

En la década de 2000, los avances en redes neuronales artificiales comenzaron a influir en el campo del NLP. Modelos como los *word embeddings* (por ejemplo, Word2Vec) y las redes neuronales recurrentes (RNN) con memoria a largo plazo (Long Short-Term Memory, LSTM) permitieron una representación más rica de las palabras y la captura de relaciones contextuales.

El lanzamiento del modelo **Transformer** en 2017, descrito en el artículo "Attention Is All You Need" por Vaswani et al [4], fue un punto de inflexión clave. El Transformer introdujo el mecanismo de atención, que permitía a los modelos enfocarse en diferentes partes del input de manera más efectiva, y mostró una gran capacidad para manejar el procesamiento paralelo de datos. Esta arquitectura fue fundamental para el desarrollo de modelos de lenguaje a gran escala. Con la introducción de BERT (Bidirectional Encoder Representations from Transformers) por Google en 2018, se popularizó la práctica de pre-entrenar modelos de lenguaje en grandes corpus de texto y luego afinarlos para tareas

específicas. BERT demostró una mejora significativa en una amplia gama de tareas de NLP al comprender mejor el contexto bidireccional.

Modelos como **GPT-2** y **GPT-3** (Generative Pretrained Transformer, por sus siglas en inglés) de OpenAI llevaron la escala de entrenamiento a un nuevo nivel, entrenándose con cientos de miles de millones de parámetros. Estos modelos demostraron una capacidad impresionante para generar texto coherente y realizar tareas de NLP sin entrenamiento específico, a través de un enfoque conocido como *few-shot learning*. El lanzamiento de GPT-4 continuó esta tendencia, con capacidades aún más avanzadas en comprensión y generación de lenguaje. Esta "nueva" forma de enfocar los LLM pronto se extenderá hasta el punto en el que nos encontramos, donde toda empresa relacionada con la tecnología tiene, de una manera u otra, un LLM personalizado propio.

2.3. Transformers: La tecnología subyacente a los LLM.

Vamos a profundizar ahora en los recién mencionados Transformers, el motor de todo lo que hemos visto hasta ahora. Los Transformers son una arquitectura revolucionaria en el campo del NLP y la IA en general. Esta arquitectura ha transformado cómo se manejan las dependencias a largo plazo en secuencias de datos, algo crucial para diversas tareas de NLP. En el artículo, los autores nos cuentan todas las innovaciones que este modelo supone con respecto a los anteriores, comparándolos con, por ejemplo, las RNNs y los LSTMs.

Los RNNs y los LSTMs, antes de la aparición de los Transformers eran los métodos más utilizados para manejar datos secuenciales en NLP. Estos modelos procesan la información de manera secuencial, paso a paso, lo que puede hacer que capturar dependencias a largo plazo en las secuencias sea un desafío debido a problemas como el desvanecimiento del gradiente. Aunque los LSTMs mitigaron parcialmente este problema con su estructura de memoria, aún tenían limitaciones significativas en términos de eficiencia y capacidad para capturar relaciones complejas a larga distancia en los datos.

La mejora más importante introducida por los Transformers es el mecanismo de atención, específicamente la atención autorregresiva. Este mecanismo permite al modelo enfocarse en diferentes partes de la secuencia de entrada simultáneamente, en lugar de procesarlas de manera lineal. La atención funciona calculando una serie de "pesos" que indican la importancia relativa de otras partes de la secuencia al generar una parte específica de la salida. Esto no solo mejora la captura de dependencias a largo plazo, sino que también permite una mayor paralelización durante el entrenamiento y la inferencia, haciendo que los Transformers sean significativamente más eficientes y escalables.

El mecanismo de atención autorregresiva se implementa mediante matrices de atención, donde cada palabra en la secuencia puede prestar atención a todas las demás palabras de manera diferenciada. Este proceso es facilitado por las "cabezas de atención múltiple", que permiten al modelo capturar diversos tipos de relaciones contextuales simultáneamente. Al utilizar múltiples cabezas de atención, el Transformer puede aprender diferentes aspectos de las relaciones en los datos de entrada, proporcionando una representación rica y matizada que es crucial para tareas complejas de NLP como la traducción automática, el resumen de textos y la respuesta a preguntas.

Además, los Transformers introdujeron el concepto de "positional encoding" para manejar la información de posición en la secuencia de datos. A diferencia de los RNNs y LSTMs, que inherentemente procesan datos en un orden específico, los Transformers no tienen una noción incorporada de secuencia. Los "positional encodings" son vectores añadidos a las entradas para proporcionar información sobre la posición relativa de cada palabra en la secuencia, permitiendo que el modelo mantenga el orden secuencial de los datos.

Otra ventaja clave de los Transformers es su capacidad para ser pre-entrenados en grandes conjuntos de datos y luego ser afinados (fine-tuned) para tareas específicas. Este enfoque de pre-entrenamiento y ajuste fino ha demostrado ser extremadamente eficaz, ya que los modelos pueden aprender representaciones profundas y generales del lenguaje durante el pre-entrenamiento, que luego se pueden adaptar de manera eficiente a tareas específicas con datos etiquetados limitados.

Pese a la increíble mejora que los Transformers supusieron para prácticamente todos los campos de la informática, siendo los favoritos a la hora de crear cualquier herramienta que vaya a tratar y procesar datos, sus cualidades técnicas no son el objetivo de este trabajo, ni los LLM en sí, como se ha indicado anteriormente, vamos a tratar solo con modelos de lenguaje que hayan sido entrenados para generar código.

No es difícil pronosticar que el exponencial crecimiento de estas herramientas y técnicas, su notable mejora frente a otras tecnologías junto con su amplio abanico de aplicaciones hace que, de manera prácticamente segura, en los próximos años veamos cómo estas herramientas siguen y siguen evolucionando y mejorando. Por ejemplo, veamos como las mejores empresas e institutos, como Gartner en este caso, sitúan a los modelos generativos de código. Pero antes vamos a conocer un poco más a Gartner.

Gartner, Inc.[9] es una destacada empresa de investigación y consultoría, reconocida mundialmente por su análisis en tecnología de la información. Fundada en 1979, Gartner se ha consolidado como una autoridad en la provisión de *insights* estratégicos, datos y asesoramiento para ayudar a líderes empresariales y tecnológicos a tomar decisiones informadas. La compañía ofrece una amplia gama de servicios, incluyendo investigación de mercado, asesoría técnica y benchmarking, y es famosa por sus informes anuales como el "Magic Quadrant" y el "Hype Cycle", que evalúan y pronostican tendencias emergentes en el sector tecnológico, aunque en este caso se hará uso de un “Impact Radar”, caracterizado por evaluar y posicionar las diversas tecnologías en función de su madurez, adopción y la magnitud de su impacto a corto y largo plazo.

Con una presencia global y una profunda experiencia en diversas industrias, Gartner se dedica a capacitar a sus clientes para que enfrenten desafíos complejos y aprovechen oportunidades en un panorama tecnológico en constante evolución, realizando un trabajo exhaustivo de investigación que resulta muy útil y es altamente respetado por las grandes empresas.

El *Impact Radar* de Gartner es una representación gráfica que no solo identifica las tecnologías emergentes, sino que también ofrece recomendaciones prácticas sobre cómo las empresas pueden prepararse y adaptarse a estos cambios. Esta herramienta es

[9] https://www.gartner.com/en

esencial para cualquier organización que busque mantenerse competitiva y relevante en un entorno tecnológico en constante evolución.

Consta de dos parámetros: "Range", que se refiere al horizonte temporal en el cual una tecnología emergente se espera que tenga un impacto significativo, se representa en diferentes anillos concéntricos, como ahora veremos, que indican el tiempo estimado para la adopción generalizada o el impacto notable de la tecnología, por lo que cuanto más cerca del centro se encuentre la tecnología, la proyección de esta es más próxima al presente. Y "Mass", que se refiere a la magnitud del impacto que una tecnología emergente puede tener en el mercado y las organizaciones, se representa por el tamaño de los puntos o burbujas que representan las tecnologías. Un mayor tamaño de la burbuja sugiere un impacto más significativo. La Figura 2 muestra el Impact Radar para las herramientas de IA generativas.

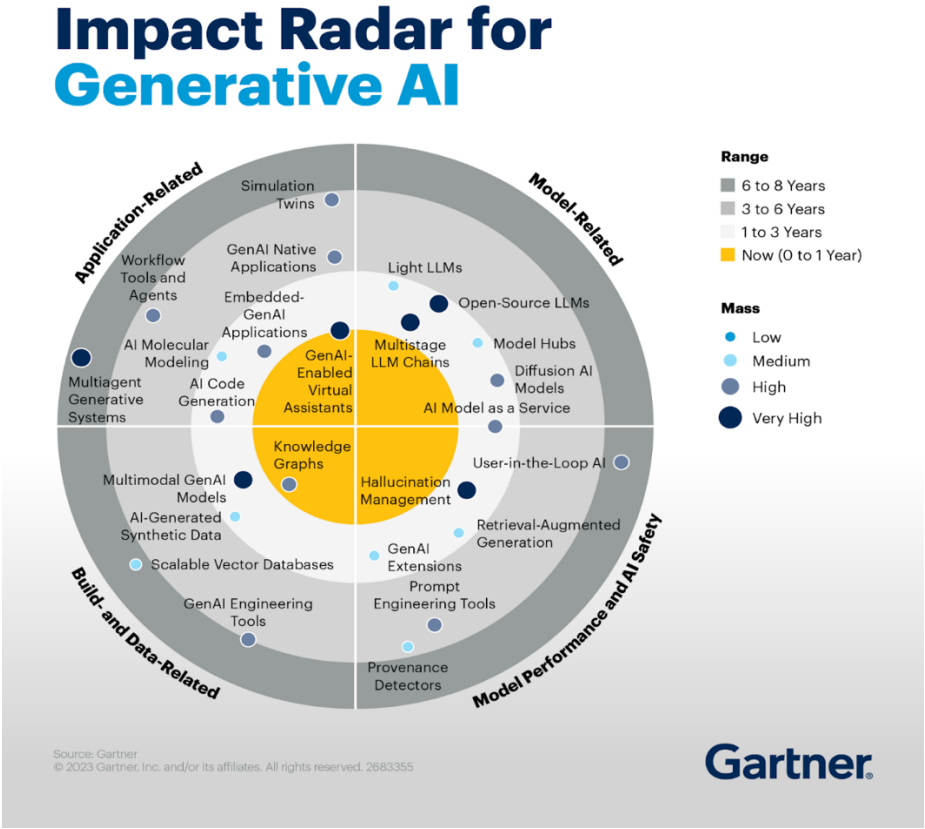


Figura 2. Impact Radar for Generative AI presentado por Gartner Inc.

Como observamos en la Figura 2, Gartner nos presenta un gráfico *Impact Radar for generative IA*, dividiendo las herramientas en 4 grandes grupos, que comentaremos empezando desde el cuadrante superior derecho y desplazándonos en sentido horario.

El cuadrante Model-*Related Innovations* agrupa a los modelos base, lo que se podría calificar como el núcleo de las ofertas de modelos generativos, junto con enfoques innovadores de estos mismos modelos, donde ya vemos que los más destacados son LLM de código abierto y librerías que combinan varios LLM para realizar tareas complejas.

Continuamos con el cuadrante *Model performance and IA safety*, donde se destaca el papel primordial del usuario en la reducción de riesgos y establecer guías para que los vendedores hagan un uso seguro de las herramientas, siendo lo más relevante el *Hallucination management*, que hace referencia a cuando el LLM proporciona respuestas muy alejadas de la realidad, como podría ser el ejemplo que hemos visto en la introducción (en la que la serpiente ha sido considerada un mamífero), aunque a una escala mucho mayor.

El cuadrante *Model build and data-related* refleja las decisiones y pasos críticos en la creación y mejora de un modelo de IA generativa, donde de nuevo destacan los *Multimodal GenIA Models*, que recogen modelos que tratan diferentes tipos de Inputs y Outputs, como imágenes, videos, etc. desde un único modelo generativo.

Y, por último, el cuadrante *IA-enabled applications*, que se centra en la expectación de aplicaciones emergentes, que darán paso a nuevos casos de uso y crearán nuevas formas de realizar tareas que ya realizamos, mejorando la experiencia de usuario. Los más destacados son los Asistentes Virtuales que se compaginan con IAs generativas, que ayudarán a estas a mejorar sus funciones y hacerlas más amigables para todos los usuarios. En este sector, y casi compartiendo con el tercer cuadrante, se encuentran los modelos generativos de código, que no tienen una masa tan grande como otros, debido a su especialización en un solo campo, pero vemos como Gartner nos indica cómo en aproximadamente 2 años esta tecnología estará completamente desarrollada.

Reforzando lo indicado en el informe Gartner, se observa que las IAs generativas de código se están desarrollando rápidamente, ofreciendo continuas mejoras en un producto que ya puede parecer muy bueno, pero que aún no estamos en un punto donde se pueda recurrir a ellas ciegamente.

2.4. Algunos ejemplos de LLM

Volviendo a los LLMs base, qué mejor manera de ilustrar su relevancia e importancia entre las grandes multinacionales tecnológicas que mostrando ejemplos de estas empresas apostando por crear herramientas de este tipo. Google y Facebook, dos gigantes en la industria de la tecnología, han desarrollado modelos de lenguaje a gran escala que están revolucionando el campo del NLP. Vamos a presentar algunos de los modelos que han desarrollado:

BERT (Bidirectional Encoder Representations from Transformers)

Google introdujo BERT en 2018, marcando un hito significativo en el desarrollo de modelos de lenguaje. BERT utiliza un enfoque bidireccional para entender el contexto de las palabras en una oración, considerando las palabras que aparecen antes y después de cada término. Esta técnica permite una comprensión más profunda y precisa del texto, lo que resulta en mejoras significativas en tareas como la búsqueda en Google, donde BERT ha sido implementado para proporcionar resultados más relevantes y precisos. Por ejemplo, BERT ha mejorado la capacidad del motor de búsqueda para entender preguntas complejas y consultas conversacionales.

T5 (Text-To-Text Transfer Transformer)

T5 es otro modelo revolucionario de Google que unifica todas las tareas de procesamiento del lenguaje bajo un enfoque de traducción de texto a texto. Esto significa que cualquier tarea de NLP, ya sea traducción, resumen, o generación de respuestas, se convierte en un problema de generación de texto. La flexibilidad y eficacia de T5 han sido demostradas en diversos benchmarks, y su enfoque unificado ha simplificado significativamente el desarrollo de aplicaciones de NLP.

XLNet

XLNet, desarrollado por Google, mejora las limitaciones de los modelos bidireccionales como BERT mediante el uso de un mecanismo de permutación. Esto permite a XLNet capturar mejor las relaciones contextuales entre palabras en un texto. Gracias a su capacidad para manejar contextos largos y complejos, XLNet ha demostrado un rendimiento superior en tareas como la respuesta a preguntas y la generación de texto. Su implementación ha sido clave en aplicaciones que requieren una comprensión profunda del contexto, como los asistentes inteligentes y los sistemas de recomendación.

RoBERTa (A Robustly Optimized BERT Approach)

RoBERTa es la versión optimizada de BERT desarrollada por Facebook IA. Al eliminar ciertas restricciones del modelo original y entrenarlo con un mayor volumen de datos durante más tiempo, RoBERTa ha logrado establecer nuevos estándares en diversos benchmarks de NLP. Las mejoras incluyen la eliminación de la tarea de predicción de la siguiente oración y el ajuste de los hiperparámetros para un rendimiento más robusto. RoBERTa se ha utilizado para mejorar la precisión de herramientas de análisis de texto, sistemas de moderación de contenido y asistentes virtuales.

BART (Bidirectional and Auto-Regressive Transformers)

Otra herramienta, y quizás la más importante de Facebook IA es BART, que combina los enfoques bidireccional y autorregresivo para la generación y comprensión de texto. BART ha demostrado ser muy eficaz en tareas de generación de texto, resumen y traducción automática. Su arquitectura permite una comprensión más profunda del contexto y una generación de texto más coherente y precisa, lo que lo convierte en una herramienta valiosa para aplicaciones que requieren la producción de texto natural de alta calidad.

3. Generadores de Código y Asistentes de Código

Los generadores de código se construyen sobre arquitecturas avanzadas de LLM como GPT (*Generative Pre-trained Transformer*), o cualquier otro modelo de lenguaje avanzado. Aunque comparten muchas características con los LLM generales, los generadores de código están específicamente optimizados y entrenados para manejar datos de código fuente. Este enfoque especializado permite que los modelos comprendan no sólo el lenguaje natural, sino también la sintaxis y semántica de múltiples lenguajes de programación. De una manera parecida a la que analizan lenguaje en distintos idiomas, el modelo entrenado es capaz de leer y entender el lenguaje escrito en "el idioma de código".

3.1. Orígenes de los Generadores de Código.

La evolución de estas herramientas está estrechamente vinculada al desarrollo de la IA y el NLP. A lo largo de las décadas, estos avances han transformado la capacidad de las máquinas para entender y generar código de manera cada vez más sofisticada, y prácticamente desde el principio, la opción de migrar los LLM para usarlos en el campo de la generación de código estuvo muy presente. A continuación, vamos a ver el recorrido histórico sobre cómo surgieron y evolucionaron los generadores de código.

Creación de lenguajes de programación de alto nivel.

Pese a que no tiene una implicación directa, un paso enorme de cara a la generación automática de código fue el salto cualitativo de los lenguajes de programación de alto nivel FORTRAN y COBOL, que nacieron en la década de 1950. Estos lenguajes sustituyeron de manera casi completa al código máquina, previamente utilizado, que permitieron que los programas se escribieran en una sintaxis más cercana al lenguaje natural. Como se ha indicado, pese no tener una relevancia directa, los LLM procesan lenguaje natural, y el hecho de que los lenguajes de programación actuales tengan características similares a este, hace que la generación de código mediante estos modelos sea más efectiva.

1980- 1990: Sistemas CASE, IDEs y MDA.

Durante la década de 1980, surgieron los primeros sistemas CASE (Computer-Aided Software Engineering) que marcaron un avance significativo en la automatización del desarrollo de software. Estas herramientas fueron diseñadas para mejorar la eficiencia y precisión del proceso de programación mediante el uso de diagramas de flujo, diagramas de entidades y relaciones, y diagramas de estructura de datos. Los sistemas CASE permitían a los desarrolladores y analistas crear representaciones visuales estructuradas de sistemas y, a partir de estas, generar automáticamente partes del código en lenguajes como COBOL, Ada y C. Sin embargo, aunque estos sistemas mejoraron la productividad y la estandarización del código, tenían una flexibilidad limitada y a menudo requerían ajustes manuales significativos para adaptarse a los cambios específicos en los requisitos del proyecto.

Herramientas como Rational Rose y Bachman CASE Tools fueron pioneras en este ámbito, proporcionando funcionalidades avanzadas para el diseño y la generación de software.

En la década de 1990, los entornos de desarrollo integrados (IDEs) comenzaron a incorporar herramientas básicas de generación de código. IDEs como Microsoft Visual Studio se convirtieron en plataformas completas para el desarrollo de software, integrando editores de código, compiladores y depuradores en una única interfaz. Estas herramientas introdujeron asistentes que permitían a los desarrolladores crear formularios e interfaces de usuario de manera visual, generando automáticamente el código necesario para la interfaz gráfica y facilitando la creación de aplicaciones de escritorio. Además, los IDEs comenzaron a incluir generadores de código *boilerplate*, que proporcionaban plantillas para tareas comunes, como la creación de clases, métodos y estructuras de datos, reduciendo el tiempo necesario para escribir código repetitivo y mejorando la consistencia y calidad del código.

Simultáneamente, la Arquitectura Dirigida por Modelos (MDA) se convirtió en una metodología popular durante esta década. El concepto clave de MDA era la creación de modelos de alto nivel que representaban los aspectos funcionales y estructurales del sistema, los cuales luego se traducían automáticamente en código ejecutable. UML (Unified Modeling Language) se estableció como el estándar para la creación de estos modelos, permitiendo a los desarrolladores diseñar diagramas de clases, diagramas de secuencia, diagramas de actividad y otros tipos de diagramas UML para representar sus sistemas.

Las herramientas de MDA podían generar automáticamente gran parte del código base a partir de los modelos UML, incluyendo la creación de clases, métodos y relaciones entre objetos, lo que aceleraba significativamente el proceso de desarrollo. Aunque MDA ofrecía beneficios claros en términos de estandarización y rapidez de desarrollo, también presentaba desafíos. La precisión del código generado dependía en gran medida de la calidad de los modelos iniciales, y cualquier cambio en los requisitos del proyecto a menudo requería ajustes manuales en el código generado. Herramientas como Rational Rose (posteriormente Rational Software Architect) y TogetherSoft ControlCenter fueron influyentes en el ámbito de MDA, proporcionando capacidades robustas para modelado y generación de código.

2000: Inicio de las herramientas de generación de código.

Hacia finales de la década de 1990 y principios de la década de 2000, las herramientas de generación de código y los IDEs continuaron evolucionando. Las herramientas de diseño comenzaron a ofrecer una integración más estrecha con los entornos de desarrollo, permitiendo iteraciones rápidas entre el modelado y la codificación. Las herramientas de generación de código comenzaron a ofrecer opciones de personalización, permitiendo a los desarrolladores definir plantillas y reglas específicas para ajustar el código generado a las necesidades particulares de sus proyectos. Este periodo sentó las bases para las sofisticadas herramientas de generación de código que conocemos hoy en día, mejorando continuamente en términos de eficiencia, precisión y capacidad de adaptación a los requerimientos cambiantes del desarrollo de software.

Todo esto nos lleva hasta hoy, donde la generación de código más avanzada y precisa reside en, como ya hemos mencionado, los modelos basados en LLM pre-entrenados para generar y entender los lenguajes de programación a un muy alto nivel. Pero vamos a centrarnos en cómo se consigue este entrenamiento y por qué da tan buenos resultados.

3.2. Cómo se pre-entrenan los generadores de código

El entrenamiento especializado de los LLM implica un proceso complejo y multifásico, diseñado para maximizar la capacidad del modelo para comprender y generar texto de manera coherente y precisa, basada en "machacar" al modelo con datos y ejemplos de código. Para cada modelo los datos empleados tienen una ligera varianza, sin embargo, los pasos a seguir son prácticamente los mismos. A continuación, se describen las etapas clave de este proceso, incluyendo ejemplos específicos de cómo se han entrenado algunos de los modelos más avanzados.

Recolección de Datos

El primer paso en el entrenamiento de un LLM es la recolección de datos a gran escala. Esta fase involucra la recopilación de grandes volúmenes de texto de diversas fuentes en Internet. Estas fuentes incluyen artículos de Wikipedia, libros, artículos de noticias, publicaciones en blogs, foros de discusión, repositorios públicos y otros textos accesibles públicamente. La diversidad de estas fuentes asegura que el modelo esté expuesto a una amplia gama de temas, estilos de escritura y contextos lingüísticos, lo que es crucial para desarrollar una comprensión profunda y versátil del lenguaje.

Preprocesamiento

Una vez recopilados los datos, se procede al preprocesamiento. Esta etapa incluye la normalización del texto, la eliminación de duplicados y la reducción del ruido. La normalización puede implicar convertir todo el texto a minúsculas, eliminar caracteres especiales o normalizar espacios y saltos de línea. La eliminación de duplicados asegura que el modelo no aprenda patrones redundantes, mientras que la reducción del ruido implica filtrar texto irrelevante o malformado que podría afectar negativamente el entrenamiento. Este paso es crucial para asegurar que el modelo reciba datos de alta calidad, lo que mejora su capacidad para aprender patrones lingüísticos significativos.

Entrenamiento Inicial

El entrenamiento inicial del modelo se realiza utilizando técnicas de aprendizaje profundo, especialmente redes neuronales profundas como los Transformers. Por ejemplo, **GPT-3** se entrenó utilizando una arquitectura de *Transformer*, de las que hemos hablado anteriormente, con aproximadamente 175 mil millones de parámetros, lo que le permitió aprender una vasta cantidad de relaciones contextuales y patrones lingüísticos. Durante esta fase, el modelo se expone a la totalidad del conjunto de datos de entrenamiento, aprendiendo a predecir la siguiente palabra en una secuencia de texto, lo que le permite

desarrollar una comprensión básica pero amplia del lenguaje. Este proceso es extremadamente intensivo en recursos y puede requerir semanas o incluso meses de computación en infraestructuras de supercomputación.

Entrenamiento Fino o Especializado

Después del entrenamiento inicial, el modelo se somete a una fase de ajuste fino. Este ajuste se realiza utilizando conjuntos de datos más específicos y especializados que permiten al modelo mejorar su rendimiento en tareas concretas. Por ejemplo, para especializar un modelo en generación de código, se puede afinar con repositorios de código fuente y documentación técnica, por ejemplo, **Amazon CodeWhisperer**[10], modelo generador de código creado por Amazon, se entrena con la documentación de AWS. Esto permite a los modelos entender mejor las estructuras de programación, la sintaxis de diversos lenguajes y las prácticas comunes en desarrollo de software. En otro ejemplo, **Replit Ghostwriter**, otro modelo generativo que veremos más adelante, utiliza datos del uso de la plataforma Replit[11] para realizar esta fase del entrenamiento.

3.3. Generadores de Código vs Asistentes de Código

Durante el trabajo hemos hablado de generadores de código en general, en este apartado vamos a hacer una diferenciación entre dos grandes grupos de herramientas generadoras de código, a las que llamaremos Generadores de Código y *Code Assistants* (o Asistentes de Código). Se explorará en detalle cómo funcionan estas herramientas, sus diferencias clave, y el impacto significativo que tienen en la productividad y calidad del código. Mientras que los generadores de código ofrecen soluciones a demandas específicas a través de interacciones conversacionales, los *Code Assistants* se integran directamente en el flujo de trabajo del desarrollador, proporcionando soporte en tiempo real.

Generadores de Código

A lo largo de este capítulo hemos hablado sobre generadores de código “planos”, herramientas basadas en LLM, diseñados para crear fragmentos de código en distintos lenguajes de programación. Estos generadores funcionan de manera similar a cómo un LLM genera texto natural: analizan una entrada y producen un resultado coherente y sintácticamente correcto en el lenguaje de código especificado.

[10] https://aws.amazon.com/es/q/developer/
[11] https://replit.com

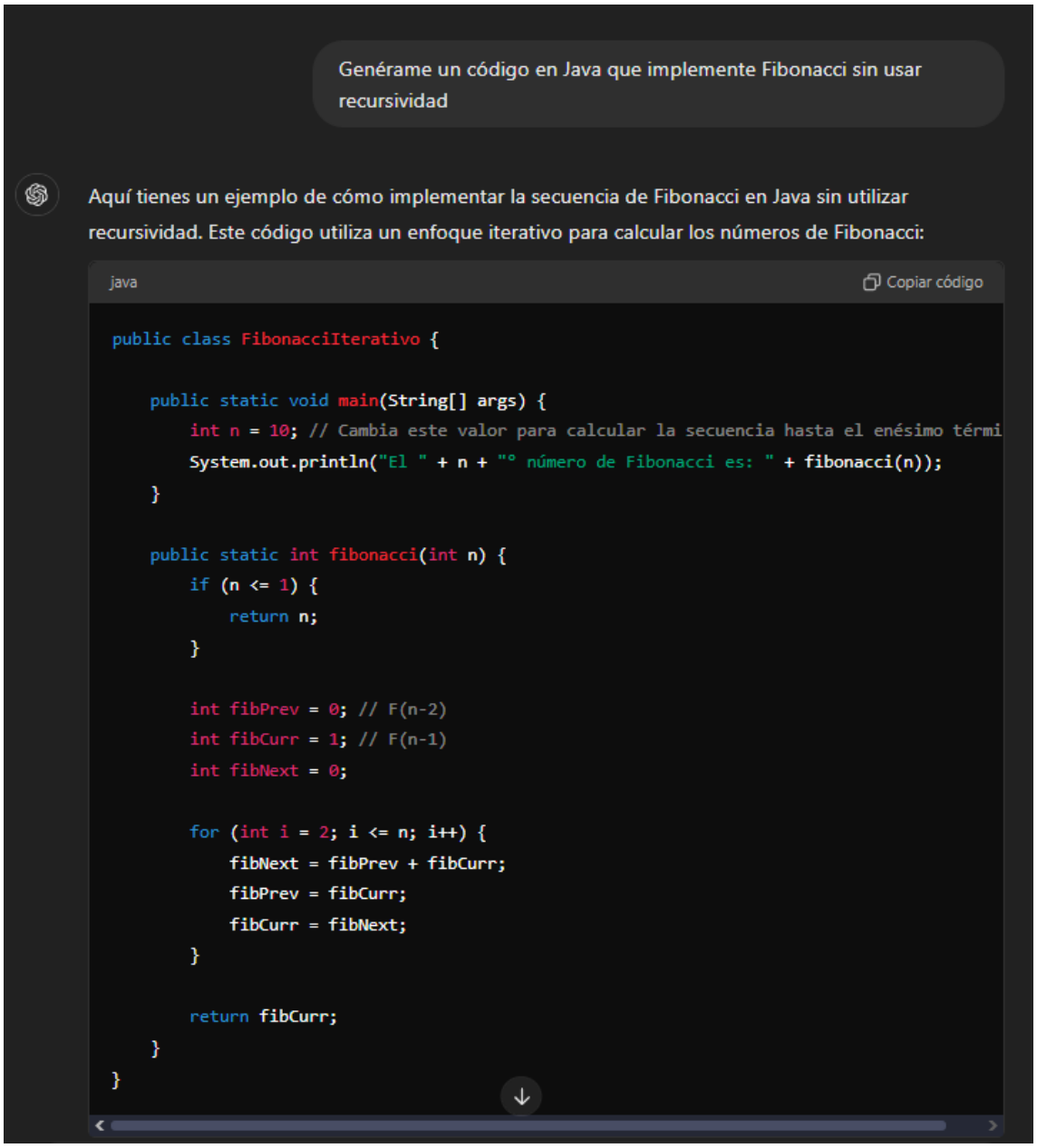


Figura 3. Ejemplo básico de código generado por ChatGPT

Como vemos en la Figura 3, el usuario proporciona a la herramienta una pequeña descripción del código que quiere generar (ya se verá más adelante cómo la interacción con la herramienta es una parte muy importante a la hora de generar un código correcto), y la herramienta, en este caso **ChatGPT**, genera el código correspondiente, a priori funcional, generando también la clase *main*, necesaria para el código en Java, una salida por consola para ver que está funcionando correctamente, y también decora el código con comentarios para que el programador pueda entender más fácilmente el código generado.

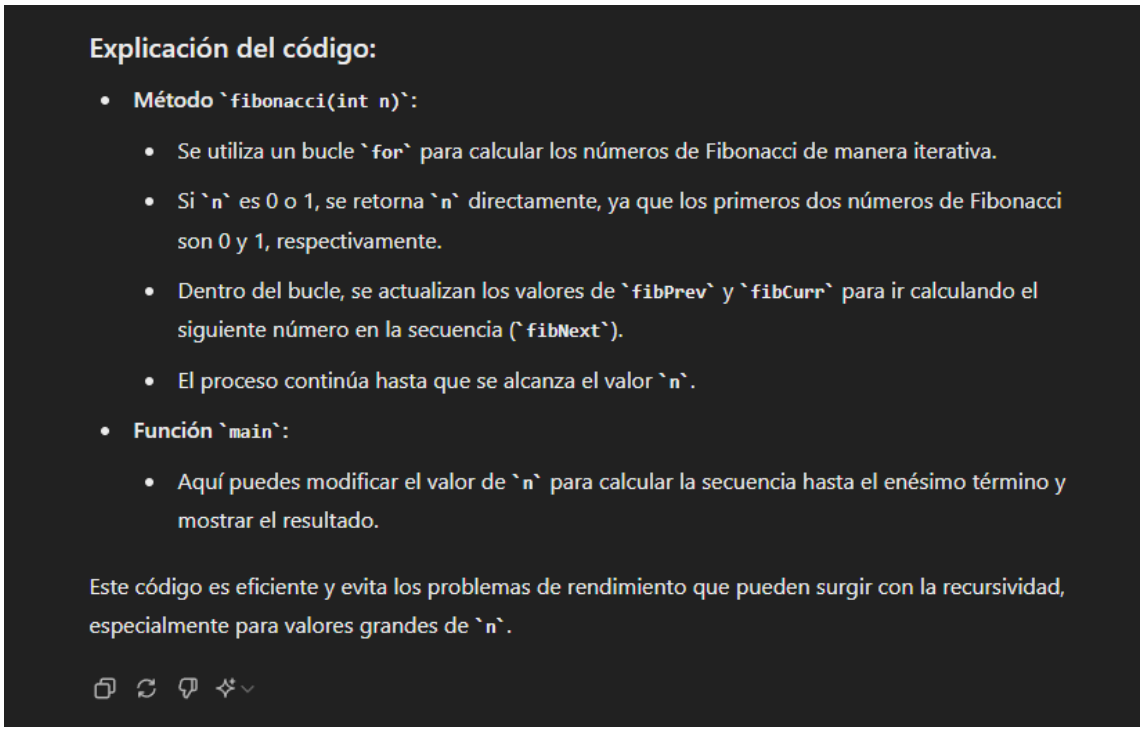


Figura 4. Explicación proporcionada por ChatGPT

Adicionalmente, estos *chatbots* cuando se usan para generar código suelen acompañar el código que proporcionan con una explicación describiendo todos los componentes usados. La Figura 4 que muestra la descripción del código generado en la Figura 3, explicando las funciones, variables dentro de cada una de ellas y cómo se manejan. Sin embargo, como se ha indicado en la introducción de este trabajo, estas descripciones no nos serán útiles (desde la perspectiva de la Ingeniería del Software) y simplemente se muestran para ver cómo funcionan estas herramientas cuando se usan para generar programas que resuelven problemas puntuales y concretos.

Así es como trabajan los generadores de código "planos": interactúas con la herramienta de manera conversacional y, a medida que describes lo que necesitas, te proporciona el código correspondiente. Este enfoque es intuitivo y eficiente, lo que permite a cualquier programador, independientemente de su nivel de experiencia, generar código robusto y completo. Por esta razón, su uso se ha extendido ampliamente en todos los niveles de programación. Sin embargo, existen otras herramientas que, aunque se basan en principios similares, operan de manera diferente: los *Code Assistants*.

Code Assitants

A diferencia de los generadores de código planos, los *Code Assistants* están diseñados para integrarse directamente en el entorno de desarrollo del programador, ofreciendo sugerencias y autocompletando fragmentos de código en tiempo real mientras el usuario escribe el código.

Estas herramientas no solo generan código en respuesta a solicitudes directas, sino que también actúan de manera proactiva al anticipar las necesidades del desarrollador. A

medida que el programador escribe, el asistente analiza el contexto del código que se está creando, comprendiendo las intenciones y patrones comunes que suelen seguirse en tareas similares. Esto le permite proponer automáticamente soluciones, sugerencias, y fragmentos de código que se alinean con lo que el desarrollador probablemente necesita en ese momento.

Por ejemplo, si el programador comienza a escribir una función en un lenguaje de programación, el asistente podría sugerir cómo completar la función basándose en su nombre y en el código circundante. Si el desarrollador está escribiendo un bucle, la herramienta podría sugerir la estructura completa del bucle, incluidas las condiciones y acciones que son típicas para ese tipo de operación. En el caso de trabajar con APIs o librerías, la herramienta puede ofrecer ejemplos de uso común o completar automáticamente las llamadas a funciones con los parámetros adecuados.

Además, estos asistentes también pueden reconocer patrones de código repetitivos y optimizar el proceso sugiriendo refactorizaciones o simplificaciones del código. Si detectan que se está escribiendo un código similar en varios lugares, pueden sugerir la creación de una función reutilizable o recomendar el uso de patrones de diseño establecidos.

Esta capacidad para anticipar y sugerir proactivamente no solo ahorra tiempo, sino que también ayuda a evitar errores comunes y a seguir las mejores prácticas de programación. En lugar de que el desarrollador tenga que detenerse para buscar cómo implementar una determinada funcionalidad o recordar la sintaxis exacta de una API, el asistente se adelanta, proporcionando las herramientas necesarias justo cuando se necesitan. Esto convierte a los *Code Assistants* en compañeros poderosos que potencian la eficiencia y la productividad en el desarrollo de software.

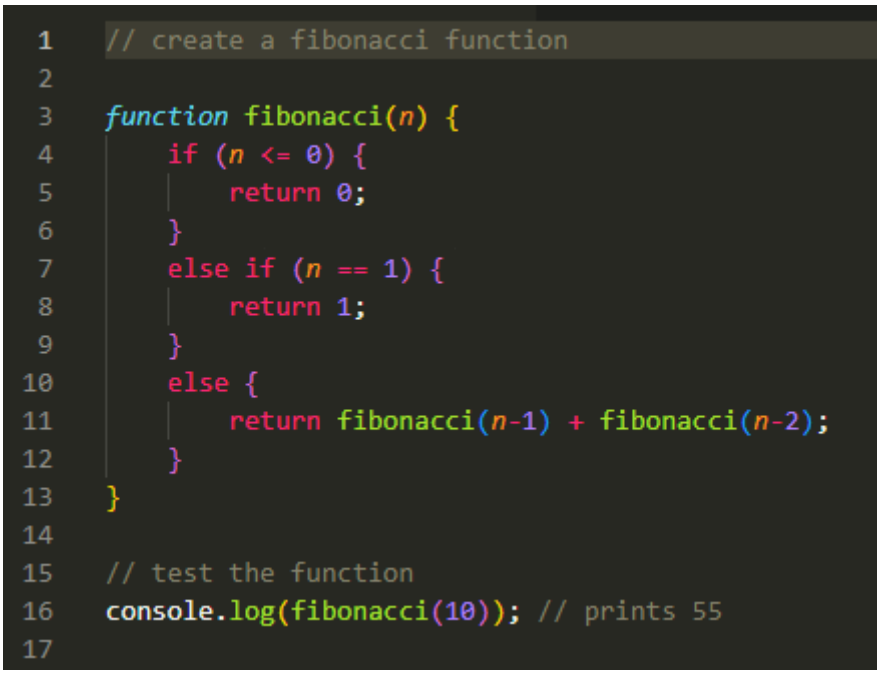
La diferencia principal con las otras herramientas generativas radica en la capacidad de los asistentes para leer y comprender el código a medida que el programador lo escribe. Esta característica ofrece una gran ventaja en términos de eficiencia, ya que elimina la necesidad de cambiar constantemente de contexto entre el entorno de desarrollo y una herramienta externa. En lugar de detenerse, formular una solicitud detallada y esperar la respuesta, el desarrollador puede recibir sugerencias y complementaciones en tiempo real, directamente en su flujo de trabajo. Esto no sólo acelera el proceso de escritura, sino que también mantiene al programador enfocado en la tarea, minimizando interrupciones y distracciones.

Sin embargo, esta inmediatez y fluidez también conlleva ciertos desafíos. Dado que los asistentes basan sus sugerencias en el análisis de fragmentos de código que aún pueden estar incompletos o en nombres de métodos que podrían ser ambiguos, no siempre logran captar con precisión la intención exacta del programador. Por ejemplo, si el nombre de un método es genérico o si el contexto del código está apenas en sus primeras etapas de desarrollo, la sugerencia del asistente podría no coincidir con lo que el programador realmente tenía en mente.

Este desajuste puede llevar a propuestas de código que, aunque sintácticamente correctas, no sean funcionales o relevantes para la tarea específica en curso. En estos casos, el desarrollador podría necesitar ajustar manualmente la sugerencia o ignorarla por completo, lo que podría mitigar parte del ahorro de tiempo que estos asistentes buscan proporcionar. Además, existe el riesgo de que los desarrolladores menos experimentados

acepten sugerencias sin una revisión crítica, lo que podría introducir errores o soluciones subóptimas en el código.

Sin embargo, al comprender sus fortalezas y limitaciones, los programadores pueden aprovechar al máximo sus capacidades, utilizándolos como una fuente de apoyo que complementa su propio juicio y conocimiento en el proceso de desarrollo, como su propio nombre indica, utilizándolos como unos asistentes.



```
// create a fibonacci function

function fibonacci(n) {
    if (n <= 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }
    else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}

// test the function
console.log(fibonacci(10)); // prints 55

```

Figura 5. Fragmento de código generado con BlackBox

Vamos a ver ahora un ejemplo de estos *Code Assistants*, en este caso **BlackBox**[12]**,** que se presenta con una versión convencional, un chat conversacional, pero también puede ser descargado como extensión en los IDEs de los usuarios para poder usarlo como asistente. BlackBox puede ser obtenido y usado de manera gratuita pero también contiene varias versiones de pago, por lo que resulta una herramienta muy polifacética.

En este caso, el modo de empleo para el ejemplo presentado en la Figura 5 ha sido muy sencillo, se ha escrito en un comentario lo que se quería hacer y línea por línea la extensión ha ido proponiendo el código, por lo que todo lo que se ha tenido que hacer es pulsar Tabulador (tecla Tab), adicionalmente también propone una prueba con la respuesta que debería de dar, a diferencia del ejemplo visto en la Figura 3.

Por otra parte, vemos que el código generado no tiene ninguna información para el programador, por lo que, tal y como se ha puntuado anteriormente, si el programador no tiene la suficiente experiencia puede ser que se vea abrumado por la propuesta de esta herramienta.

[12] https://www.blackbox.ai

En resumen, las herramientas de generación de código basadas en chat conversacional, como ChatGPT, y los *Code Assistants* integrados en IDEs, como BlackBox, han transformado la programación al ser altamente accesibles y fáciles de usar. Los asistentes conversacionales permiten a los desarrolladores interactuar en lenguaje natural, obteniendo respuestas en tiempo real, lo que facilita el aprendizaje y la resolución de problemas sin necesidad de consultar múltiples fuentes.

Por su parte, los asistentes en IDEs ofrecen sugerencias de código, autocompletado y detección de errores directamente en el entorno de desarrollo. Esto mejora la calidad del código y acelera el proceso, sin requerir configuraciones complejas o una curva de aprendizaje pronunciada, integrándose de manera fluida en el flujo de trabajo diario. Ambos tipos de herramientas.

3.4. ¿Por qué pre-entrenar LLMs para generar código?

Anteriormente en el trabajo se ha hablado de cómo los LLM generan código de la misma manera que lo hacen con texto normal, pese a que técnicamente es cierto, no requiere la misma complejidad para estas herramientas saber escribir y leer en muchos idiomas que la que requiere crear un fragmento de código como lo haría un desarrollador experto. Por esto es por lo que los modelos conversacionales "normales", pese a que tienen la capacidad de generar código, nunca podrán hacerlo de una manera profesional.

Es por esto por lo que el pre-entrenamiento de modelos de lenguaje es un paso crucial para crear versiones que puedan generar código de manera robusta y eficiente. Esto se debe a que los modelos de lenguaje, como GPT-3 y OpenAI Codex, necesitan una base sólida de conocimientos y habilidades antes de poder manejar la complejidad y precisión requeridas en la generación de código. Estos modelos no solo deben interpretar comandos en lenguaje natural, sino también transformar esos comandos en código ejecutable que siga estándares de calidad y buenas prácticas en la programación.

Para entender el porqué es necesario pre-entrenar estos modelos, primero es importante reconocer que el código, como hemos dicho, no es simplemente texto común. Es una forma de comunicación altamente estructurada con reglas gramaticales específicas (la sintaxis), significado (la semántica) y un contexto funcional que debe ser comprendido y seguido rigurosamente para que el código sea ejecutable y cumpla su propósito. Además, a diferencia del lenguaje natural, donde a veces se puede permitir ambigüedad o errores menores, el código requiere una precisión absoluta: incluso un pequeño error, como una coma fuera de lugar o un nombre de variable incorrecto, puede causar que un programa falle o se comporte de manera impredecible.

Un modelo de lenguaje que genera código debe cumplir con varias exigencias complejas para ser verdaderamente útil, ya que como se ha puntuado generar código no es una tarea sencilla, y generar un código bueno y robusto requiere de mucho conocimiento detrás. Vamos a ver los principales puntos que debe dominar un modelo de lenguaje capacitado para generar código:

Comprender la sintaxis de múltiples lenguajes de programación

A diferencia del lenguaje natural, donde el significado puede interpretarse incluso con errores gramaticales, el código debe ser exacto. Una sola coma fuera de lugar puede hacer que un programa falle. Por lo tanto, el modelo debe tener una comprensión profunda de las reglas sintácticas de los lenguajes de programación en los que está entrenado. Esto incluye no sólo la gramática del lenguaje, sino también los distintos paradigmas de programación, como la programación orientada a objetos, funcional o procedimental.

Conocer patrones comunes y mejores prácticas

El código no solo necesita funcionar, sino que también debe ser eficiente, seguro y fácil de mantener. Esto implica seguir patrones y prácticas que son comunes en la programación, algo que el modelo debe aprender. Por ejemplo, debe entender cómo manejar excepciones correctamente, cómo optimizar el rendimiento del código y cómo estructurar el código para que sea legible y mantenible. El pre-entrenamiento en grandes bases de datos de código fuente permite al modelo internalizar estos patrones y aplicarlos de manera automática cuando genera nuevo código.

Aplicar lógica y contexto

La generación de código requiere una comprensión del contexto en el que se está utilizando. Por ejemplo, si un usuario solicita una función para ordenar una lista, el modelo debe saber qué algoritmos de ordenación son más apropiados para diferentes escenarios y cómo implementarlos correctamente. Además, debe ser capaz de mantener la coherencia lógica en el código, asegurándose de que todas las partes del programa trabajen juntas sin conflictos. Esto significa que el modelo necesita no solo generar código correcto en términos sintácticos, sino también entender cómo ese código se integrará con el resto del proyecto en el que está trabajando el desarrollador.

Todo esto, pese a ser muchos conocimientos, es una pequeña parte de lo que hay que saber para generar el mejor código posible, las librerías más actualizadas, lo necesario en el contexto de tu aplicación o incluso el lenguaje que has de utilizar para los fines de tu Software son factores que requieren de mucho conocimiento en el campo

El pre-entrenamiento es el proceso mediante el cual estos modelos son expuestos a vastas cantidades de datos, tanto de texto general como de código fuente, para que puedan aprender las reglas, patrones y contextos necesarios para generar código útil y funcional. Durante este proceso, el modelo no solo aprende a manejar el lenguaje natural, sino que también adquiere una comprensión profunda de cómo se escribe y estructura el código en la práctica. Este conocimiento previo es lo que permite a los modelos como Github Copilot u OpenAI Codex generar código que no solo funciona, sino que también es de alta calidad y se ajusta a las expectativas y necesidades del usuario.

Este enfoque de pre-entrenamiento especializado es esencial para que los modelos puedan manejar la complejidad del desarrollo de software, permitiendo que sean herramientas efectivas para los programadores y no simplemente generadores de texto que ocasionalmente producen código.

3.5. Integración en el desarrollo del Software

El desarrollo de software es un proceso complejo y multifacético que abarca varias etapas críticas, cada una de las cuales es esencial para la creación de aplicaciones robustas, eficientes y adaptadas a las necesidades específicas de los usuarios. Este proceso, mostrado en la Figura 6, comienza con la planificación, donde se definen los requisitos, se establecen los objetivos del proyecto y se diseña la arquitectura del sistema. A continuación, se lleva a cabo la codificación, donde los desarrolladores traducen las especificaciones en un lenguaje de programación que las máquinas pueden interpretar.

La prueba es la siguiente etapa crucial, en la que se verifica que el software funcione correctamente y cumpla con los requisitos establecidos, identificando y corrigiendo errores o inconsistencias en el código. Una vez que el software ha pasado por las pruebas, se procede a la implantación, donde se despliega en un entorno de producción para su uso real. Finalmente, el mantenimiento, quizás la más tediosa, implica realizar mejoras continuas, corregir errores y adaptar el software a nuevas necesidades o cambios en el entorno tecnológico.

Figura 6. Representación del ciclo de vida del desarrollo del software realizada por Innevo

Tradicionalmente, este proceso ha requerido una profunda comprensión de diversos lenguajes de programación, estructuras de datos, algoritmos y patrones de diseño. Los desarrolladores necesitan no solo habilidades técnicas avanzadas, sino también una gran atención a los detalles y la capacidad de adaptarse constantemente a nuevos desafíos, como la evolución de las tecnologías y las demandas cambiantes del mercado. Además, deben ser capaces de integrar nuevas funcionalidades sin comprometer la estabilidad y rendimiento del software, lo que exige un enfoque meticuloso y un conocimiento sólido de las mejores prácticas en ingeniería de software, así como una completa comprensión de todo el código que la aplicación contiene, así como cómo este interactúa entre sí.

Este conjunto de habilidades, combinado con la experiencia y el juicio crítico, ha sido fundamental para el éxito en el desarrollo de software a lo largo de los años, haciendo que el rol de desarrollador de Software “transcienda” de la de un desarrollador, siendo alguien capacitado para generar aplicaciones desde cero de una manera sólida y experta.

Vemos como el uso de las herramientas de IA afecta significativamente no solo en la fase de codificación, sino que a lo largo de todo este proceso las herramientas pueden adoptar un rol muy importante.

Planificación y Diseño

En la fase de planificación y diseño, estas herramientas pueden ayudar a los desarrolladores a diseñar y estructurar mejor sus proyectos. Los generadores de código con el modelo de *chatbots* pueden facilitar la creación rápida de prototipos al generar código inicial basado en descripciones de proyectos de alto nivel.

Esto permite a los equipos explorar diferentes enfoques arquitectónicos y obtener información sobre cómo se pueden implementar ciertas características. Además, pueden proporcionar plantillas de diseño que se alineen con las mejores prácticas de la industria, guiando a los desarrolladores a crear una base sólida para sus proyectos. Reduciendo drásticamente el tiempo que estos emplean en generar o pensar toda la estructura base del proyecto, que sabemos que es un apartado muy complejo y del cual depende el resto del desarrollo

Codificación

En la fase de codificación, el impacto de estas herramientas es quizás el más evidente. Los *Code Assistants*, integrados directamente en los entornos de desarrollo, ofrecen autocompletado inteligente, sugerencias en tiempo real y optimización del código a medida que el desarrollador escribe. Esto no sólo acelera el proceso de codificación, sino que también reduce la probabilidad de errores, asegura la adherencia a las mejores prácticas y permite a los desarrolladores concentrarse en la lógica, haciendo que las prestaciones del código sean mejores.

Además, como hemos visto, los generadores de código conversacionales son de nuevo un apoyo enorme para los programadores a la hora de realizar tareas repetitivas (como por ejemplo la creación de clases), o buscar soluciones a pequeñas tareas que se puedan resolver fácilmente.

Pruebas o Testing

En la fase de pruebas, los generadores de código "planos" pueden ser utilizados para generar automáticamente pruebas unitarias y casos de prueba basados en el código existente. Esto asegura que el software esté adecuadamente cubierto por pruebas, lo que es crucial para la calidad y estabilidad del producto final. Con un buen uso de estas herramientas podemos generar una batería de casos de prueba muy extensa que nos ayude a comprobar que lo que hemos desarrollado sea correcto.

Los *Code Assistants* también pueden detectar áreas del código que podrían beneficiarse de pruebas adicionales, mejorando así la cobertura y reduciendo la posibilidad de errores no detectados, como hemos visto en la Figura 5, donde la herramienta nos proponía pruebas para comprobar el correcto funcionamiento del código.

Implantación

Durante la implantación, la IA puede automatizar muchas de las tareas relacionadas con el despliegue del software. Las herramientas basadas en IA pueden ayudar a optimizar los scripts de despliegue y asegurar que el proceso sea lo más fluido y libre de errores posible. Esto reduce significativamente el tiempo necesario para llevar el software desde el entorno de desarrollo hasta el entorno de producción.

Aunque posiblemente sea el campo donde menos pueda ayudar, dado que no es un proceso fijo, y cada grupo de trabajo puede tener una manera distinta de realizar este procedimiento, existen infinidad de maneras de realizar el despliegue de una aplicación, quizás no haya una correcta, pero la IA puede simplemente guiarnos y ayudarnos si no tenemos mucha experiencia en esta fase.

Mantenimiento

De una manera parecida a la fase de codificación, en la fase de mantenimiento estas herramientas influyen de la misma manera, dado que una parte de esta fase es codificar en sí, pero la fase de mantenimiento no es solo eso, hay un gran número de tareas y funciones en las que nos podemos apoyar en estas herramientas para aligerar la carga de trabajo que supone realizar un buen mantenimiento.

Pero no solo en esta fase se realizan nuevas funciones de nuestro software, sino que además el software debe adaptarse a nuevas necesidades, corregir errores que han surgido en el uso real, y mejorar su rendimiento y funcionalidad para seguir siendo relevante y eficiente.

Estas herramientas también pueden realizar análisis estáticos y sugerir optimizaciones que no son evidentes a simple vista, como la mejora en la gestión de recursos, la eliminación de cuellos de botella en el rendimiento o la corrección de vulnerabilidades de seguridad. Al identificar y proponer estas mejoras de manera automatizada, los desarrolladores pueden mantener la calidad del software sin tener que recurrir a extensas revisiones manuales, lo que es especialmente beneficioso en proyectos grandes y complejos.

En el contexto de mantenimiento continuo, donde el software debe evolucionar para mantenerse relevante y competitivo, la capacidad de estas herramientas para automatizar tareas repetitivas, optimizar el código existente y prever problemas potenciales se convierte en un recurso inestimable. No solo ayudan a prolongar la vida útil del software, sino que también aseguran que se mantenga alineado con las expectativas de rendimiento y calidad, incluso a medida que cambian las demandas del mercado o surgen nuevas tecnologías.

En resumen, al integrar Code Assistants y generadores de código "planos" en la fase de mantenimiento, los desarrolladores pueden gestionar mejor la complejidad del software a largo plazo, garantizando su sostenibilidad y adaptabilidad en un entorno tecnológico en constante evolución.

4. El Prompt en el uso de los LLM

Hemos visto ya qué son los LLMs y sus especificaciones en modo de generadores de código, pero estos no serían nada sin una apropiada interacción con un programador. Como hemos resaltado en los ejemplos vistos en los capítulos anteriores, no solo tiene que haber un experto controlando el código generado, sino que también tiene que saber cómo interactuar con la herramienta.

4.1. ¿Qué es el Prompt?

A esta interacción por parte del programador, se le conoce como Prompt, que es, esencialmente, la instrucción o entrada que se le proporciona al modelo para guiar su respuesta o salida. Actúa como una especie de pregunta o punto de partida que define lo que el modelo debe generar, en nuestro caso, una pequeña explicación de lo que debe realizar el código, normalmente unido a un conjunto de especificaciones técnicas, que pueden indicar el lenguaje, la estructura o técnicas que tiene que seguir, o algo tan simple como el nombre de las variables. Como hemos visto en la Figura 5, se le pide al modelo un código que implemente Fibonacci, pero con las especificaciones de que sea en Java y que no emplee recursividad, por lo que estamos indicando exactamente lo que queríamos, de manera que dejamos menos elecciones a la herramienta.

Pese a que pueda parecer algo sencillo, dado que el ejemplo lo es, la calidad y precisión del Prompt son cruciales porque influyen directamente en la relevancia, coherencia y utilidad de la respuesta que el modelo generará. Un Prompt bien formulado puede producir resultados altamente satisfactorios, mientras que un Prompt vago o ambiguo puede dar lugar a respuestas menos útiles o fuera de contexto. Esto ha dado lugar a una nueva disciplina denominada “Ingeniería del Prompt” (*prompt engineering*) por su relevancia para la interacción con modelos de inteligencia artificial basados en lenguaje natural [5].

En el contexto del desarrollo de software, los Prompts no solo se usan para generar código, sino que, al emplear modelos generativos, los desarrolladores pueden utilizar Prompts para automatizar, optimizar y facilitar varias actividades críticas del proceso de desarrollo, lo que permite trabajar de manera más eficiente y con mayor precisión. Un buen Prompt puede dirigir al modelo a generar código que sea más eficiente, seguro, y que cumpla con los requisitos del proyecto.

Esto hace que los Prompts sean herramientas poderosas para mejorar la productividad y la calidad en el desarrollo de software. La capacidad de utilizar Prompts de manera efectiva se está convirtiendo en una habilidad crucial para los desarrolladores, dado que estos modelos de IA continúan evolucionando y desempeñando un papel más central en la ingeniería de software.

4.2. Algunos estudios

Es por esto por lo que los esfuerzos por parte de los desarrolladores y personas del mundo del Software para investigar e intentar entender a fondo cómo explotar el Prompt hayan crecido exponencialmente en los últimos años. Vamos a ver algunos de ellos.

BIG-Bench

El proyecto BIG-Bench *(Beyond the Imitation Game Benchmark*) [6] es un esfuerzo colaborativo a gran escala diseñado para evaluar y comprender las capacidades y limitaciones de los modelos de lenguaje avanzados. Contribuido por 450 autores de 132 instituciones, entre ellas la UPV, BIG-Bench incluye 204 tareas que cubren una amplia gama de temas, desde la lingüística y el desarrollo infantil hasta la física, la biología y la programación de software.

BIG-Bench es especialmente relevante en el contexto actual de la inteligencia artificial debido a su enfoque en el desarrollo y la evaluación de modelos de lenguaje a través de prompts o instrucciones que guían las respuestas de estos modelos. Un Prompt bien diseñado puede revelar la profundidad del entendimiento del modelo, su capacidad para razonar de manera lógica, interpretar contexto, manejar ambigüedades y adaptarse a distintas situaciones. En este sentido, el uso de prompts en BIG-Bench no solo sirve para medir la precisión de las respuestas generadas, sino también para explorar las formas en que los modelos responden a diferentes tipos de instrucciones, lo que resulta crucial para identificar sesgos, limitaciones y potenciales mejoras.

La iniciativa BIG-Bench se distingue por su enfoque en la colaboración abierta, invitando a investigadores de todo el mundo a proponer nuevas tareas y contribuir al análisis de resultados. Este enfoque colaborativo no solo amplía el conjunto de datos y tareas disponibles, sino que también fomenta una comunidad dinámica de investigadores que trabajan juntos para avanzar en el campo de la inteligencia artificial.

Improving ChatGPT Prompt for Code Generation [7]

Este estudio se enfoca en dos tareas clave de generación de código: Text-to-Code (T2C), donde se genera código a partir de descripciones en lenguaje natural, y Code-to-Code (C2C), donde se traduce código de un lenguaje de programación a otro. Los autores utilizaron el modelo ChatGPT, basado en GPT-3.5, para llevar a cabo estas tareas, utilizando el conjunto de datos CodeXGlue como referencia.

Los resultados muestran que el rendimiento de ChatGPT en la generación de código mejora significativamente cuando se diseñan prompts específicos y optimizados. El estudio sugiere utilizar una estrategia de Chain-of-Thought (CoT), que guía a ChatGPT a través de pasos intermedios antes de generar la respuesta final, lo que mejora la precisión y calidad del código generado.

Que esta técnica, que en resumidas cuentas consiste en ir “guiando” al modelo para modelar la respuesta final que el usuario desea, sea la más eficiente respalda lo que se ha puntuado durante todo el trabajo, que estos modelos, pese a ser muy poderosos, necesitan a los humanos para funcionar correctamente.

Benchmarks

Antes de empezar, vamos a hablar de cómo los autores de dichos trabajos establecen una serie de métricas y objetivos para llevar estos a cabo, lo que se conoce como *Benchmarks*, que se han nombrado anteriormente pero solo de pasada.

En el contexto de generación de código, un *Benchmark* es un conjunto de pruebas estandarizadas diseñadas para evaluar y comparar el rendimiento de modelos de lenguaje o sistemas de inteligencia artificial en tareas específicas relacionadas con la escritura de código. Estos *Benchmarks* suelen incluir problemas de programación que los modelos deben resolver, y su desempeño se mide en términos de precisión, eficiencia y capacidad para producir soluciones correctas y funcionales.

Los *Benchmarks* en este contexto no solo evalúan la capacidad del modelo para generar código sintácticamente correcto, sino también para producir soluciones que cumplen con los requisitos de funcionalidad, eficiencia en el uso de recursos (como tiempo de ejecución y memoria), y capacidad de adaptarse a diferentes tipos de problemas de programación.

Estos *Benchmarks* son esenciales para determinar la efectividad de un modelo en aplicaciones prácticas de desarrollo de software, y cada autor crea y personaliza el *Benchmark* que quiera emplear en base a la naturaleza del estudio.

HumanEval Dataset

Si hablamos de Prompts, y ya habiendo introducido el concepto de *Benchmark* qué mejor manera de plasmar su importancia que viendo la relevancia de unos de los *Datasets* más usados cuando se habla de estudiar estas herramientas, **HumanEval**.

En términos generales, HumanEval Dataset es un conjunto de datos creado por OpenAI en 2021 que se utiliza para medir el rendimiento de modelos de lenguaje natural en la tarea de generación de código, conocido como *Benchmark*, con especial atención en el lenguaje de programación Python, pero aplicable a cualquiera realmente. Este dataset se diseñó específicamente para evaluar la capacidad de los modelos de completar funciones de código, basándose en descripciones textuales o comentarios que especifican lo que debe hacer la función.

Consiste en una colección de 164 problemas de programación de diversa dificultad. Cada problema viene con una descripción en lenguaje natural, una función parcialmente completada (o esqueleto de código) y pruebas unitarias predefinidas (como veremos más adelante en este capítulo). Las pruebas unitarias son esenciales, ya que se utilizan para verificar si el código generado por el modelo es correcto y cumple con los requisitos especificados en la descripción.

Este conjunto de problemas estandarizado permite comparar diferentes modelos de lenguaje en términos de sus capacidades de programación de una manera unificada, cubriendo una variedad de tareas comunes en la programación y permitiendo evaluar la capacidad de los modelos para generalizar y resolver diferentes tipos de problemas.

Por todo esto, HumanEval Dataset se ha convertido en un estándar clave en los estudios sobre generación de código, especialmente para evaluar el rendimiento de los modelos de lenguaje en esta tarea. Desde su creación en 2021, ha sido utilizado en una amplia variedad de investigaciones y desarrollos de inteligencia artificial, tanto en la industria como en el ámbito académico.

Codex (OpenAI)

Codex, el modelo detrás de GitHub Copilot, fue uno de los primeros modelos en ser evaluado usando HumanEval, cómo puede resultar obvio, dado que OpenAI creó este Dataset. En el estudio de OpenAI, se probó la capacidad del modelo para generar código funcional a partir de descripciones textuales, estableciendo un *Benchmark* inicial en el que otros modelos han sido comparados.

PaLM (Google)

El modelo PaLM, desarrollado por Google, también fue evaluado utilizando HumanEval. En este contexto, PaLM demostró una fuerte capacidad en la generación de código, aunque el modelo LLaMA-65B, más tarde logró superarlo en este mismo *Benchmark*.

task_id

```
HumanEval/0
```

prompt

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

canonical_solution

```
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

test

```
METADATA = {
    'author': 'jt',
    'dataset': 'test'
}

def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
```

Figura 7. Representación del problema con ID: 0 de HumanEval Dataset [8]

En la Figura 7 vemos representado de manera gráfica un ejemplo de un problema que propone HumanEval, donde se pueden observar claramente las partes fundamentales del *dataset* que hemos mencionado anteriormente. En primer lugar, se presenta el *Prompt*, que actúa como el enunciado del problema. En esta sección, se especifica la cabecera del método que el modelo debe generar, indicando claramente qué tipos de datos recibe como parámetros y qué tipo de datos debe devolver como resultado. Además de esto, el Prompt incluye una breve explicación del funcionamiento que el método debe implementar, proporcionando contexto sobre lo que se espera que el código haga. Esto a menudo incluye ejemplos de entradas y salidas esperadas, lo que ayuda al modelo a comprender mejor las reglas y patrones necesarios para resolver el problema correctamente.

A continuación, se encuentra el apartado de la *canonical_solution*, que es un componente crítico del problema. Esta es la solución escrita por un programador humano que resuelve el problema propuesto de manera eficiente y correcta. Esta solución se considera la referencia ideal contra la cual se mide la calidad del código generado por el modelo. La solución canónica actúa como una guía en la evaluación de la precisión del modelo, asegurando que el código generado no solo sea funcional sino también comparable en calidad a una solución escrita manualmente.

Finalmente, se incluye el apartado de *test*, que es una lista de pruebas unitarias diseñadas para determinar si el código generado cumple con las especificaciones funcionales descritas en el Prompt. Estas pruebas unitarias son esenciales para la evaluación automática del código, ya que permiten verificar si el modelo ha entendido correctamente la tarea y si el código que ha generado funciona en todos los casos previstos. Las pruebas están diseñadas para cubrir tanto los casos comunes como aquellos extremos, asegurando una validación completa del código.

Model	Base Model	Size	Year	Benchmark	Score
GPT-NEO [18]	GPT-2	125M 1.3B 2.7B	2021	HumanEval	02.97 16.30 21.37
GPT-J [19]	GPT-2	6B	2021	HumanEval	27.74
Codex [20]	GPT-3	300M 679M 2.5B 12B	2021	HumanEval	36.27 40.95 59.50 72.31
TabNine	?	?	2021	HumanEval	7.59
ChatGPT	GPT-3.5	?	2022	HumanEval	94.00
Gemini-Ultra [17]	Transformer	?	2023	HumanEval Natural2Code	74.40 74.90
Gemini-Pro [17]	Transformer	?	2023	HumanEval Natural2Code	67.70 69.60
CodeGen [21]	Auto-regressive Transformer	350M 2.7B 6.1B 16.1B	2023	HumanEval	35.19 57.01 65.82 75.00
SantaCoder [22]	Decoder	1.1B	2023	MultiPL-E	45.90
InCoder [23]	Transformer	1.1B 6.7B	2023	HumanEval	25.20 45.00
StarCoder [12]	StarCoder-Base	15.5B	2023	HumanEval MBPP	33.60 52.70

Figura 8. Comparación de Generadores de Código + Benchmarks utilizados [9]

La Figura 8 presenta una tabla que recoge resultados de estudios realizados sobre las herramientas indicadas en la columna de la izquierda. Lo interesante es observar la columna de "*Benchmark"*, donde vemos una clara mayoría por parte de HumanEval, ya que, como se ha recalcado, su variedad y complejidad hacen de este DataSet uno de los mejores para cualquier clase de estudio.

Esto nos permite destacar la relevancia crucial del Prompt en el estudio de la generación de código, ya que, como hemos dicho, juega un papel fundamental en el proceso. Al observar este dataset, que se emplea tanto en el testeo como en la implementación de herramientas de gran importancia, podemos apreciar cómo se presta especial atención a la formulación del Prompt. Este se elabora con un alto nivel de detalle y precisión, proporcionando una descripción exhaustiva del problema, lo cual facilita que las herramientas puedan interpretarlo correctamente y generar código de calidad acorde a las expectativas.

5. Análisis de distintas herramientas

Ahora que ya sabemos cómo emplear estas herramientas de manera eficiente a través del Prompt, es importante profundizar en las herramientas seleccionadas para su estudio, destacadas no solo por su funcionalidad y capacidades, sino también por la magnitud de las empresas que las desarrollaron y la naturaleza innovadora de sus aplicaciones. Estas herramientas representan el fruto de años de investigación y desarrollo en inteligencia artificial, procesamiento de datos, y automatización, respaldadas por gigantes tecnológicos que lideran la industria, y como son las empresas más grandes, serán las que presenten las mejores herramientas en la mayoría de los casos .

Aunque, como ya hemos mencionado, en el mercado actual existe una gran variedad de herramientas similares, muchas de ellas basadas en principios y tecnologías comunes, presentan diferencias significativas debido a pequeñas modificaciones, mejoras y adaptaciones específicas. Esto permite que cada una tenga su propio nicho de mercado y se adapte a diferentes necesidades y objetivos.

Por lo que, a pesar de la abundancia de alternativas, las herramientas que examinaremos a continuación han logrado sobresalir en el mercado no solo por la reputación de las compañías que las desarrollan, sino también por su capacidad para innovar y liderar en el ámbito de la tecnología avanzada, sentando un precedente en la forma en que interactuamos con la inteligencia artificial y los sistemas automatizados.

5.1. OpenAI Codex[13]

Para empezar, veremos posiblemente la empresa de IA más conocida actualmente, OpenAI, de la cual ya hemos hablado numerosas veces en este trabajo, pero ahora nos centraremos en su herramienta orientada a la generación de código, pre-entrenada como ya hemos visto, y que se ha convertido en la base para otras herramientas muy importantes, como una que veremos más adelante, Github Copilot.

Codex es una extensión de GPT-3, el LLM desarrollado por OpenAI. Si bien GPT-3 es conocido por generar texto coherente en lenguaje natural, e incluso código, Codex va un paso más allá al aplicar estas capacidades a un lenguaje de programación más preciso y experto. Es capaz de trabajar con múltiples lenguajes de programación, como Python, JavaScript, Ruby, Swift, entre muchos otros.

Cabe destacar que OpenAI ofrece acceso a Codex a través de su API en un modelo de pago basado en el uso. Los desarrolladores y empresas que deseen utilizar Codex pueden acceder a la API mediante suscripción, con precios que varían según el volumen de uso. El modelo de precios se basa en el número de "tokens" utilizados, donde un token representa una pequeña porción de texto procesado por el modelo, lo que incluye tanto las solicitudes enviadas como las respuestas generadas por Codex.

[13] https://openai.com/index/openai-codex

Origen y desarrollo

OpenAI Codex se presentó públicamente en agosto de 2021 y, como otros modelos de OpenAI, fue entrenado utilizando grandes volúmenes de datos disponibles en internet, que incluían enormes cantidades de código fuente de diferentes lenguajes, obtenidos de repositorios públicos y tutoriales en línea, pero también material de documentación técnica, libros de programación y recursos educativos. El modelo se nutrió de estos datos y de la comprensión del lenguaje natural para aprender a escribir código que no sólo funcione, sino que también sea legible y eficiente.

Pre-Entrenamiento

Como se ha mencionado, el pre-entrenamiento de Codex se basa en el uso de grandes cantidades de datos de código fuente, repositorios públicos de GitHub, libros de texto sobre programación, foros de desarrolladores, y otros recursos abiertos. Esto permite a Codex "aprender" a escribir código en diversos lenguajes de programación, interpretar código existente, identificar errores y hasta optimizar algoritmos.

Dado que se entrena en una base de datos gigantesca de código y documentación, Codex no solo es capaz de escribir código, sino que también puede explicar el propósito de bloques de código, explicando al usuario cada fragmento de código que ha generado, sugerir mejores prácticas o adaptar código existente a nuevas funcionalidades. Lo que hace de esta herramienta no solo un gran compañero para generar código, sino un gran profesor también.

¿Cómo se usa?

Codex se usa principalmente a través de la API de OpenAI[14], que permite interactuar con el modelo de manera programática mediante solicitudes HTTP. A través de esta API, los desarrolladores pueden enviar instrucciones en lenguaje natural, y Codex responde generando el código correspondiente en tiempo real. Esta interacción permite a los desarrolladores transformar sus ideas o problemas expresados en lenguaje cotidiano en soluciones programáticas, sin la necesidad de escribir manualmente cada línea de código.

La facilidad de uso es notablemente flexible. Por ejemplo, un programador puede pedir algo tan simple como la creación de una función en Python que sume dos números, y Codex no solo genera la función correcta, sino que también puede proporcionar comentarios explicativos o variaciones optimizadas. Al mismo tiempo, la API también puede manejar tareas mucho más complejas, como diseñar y escribir módulos completos para una aplicación web desde cero, gestionar bases de datos, o incluso implementar interfaces de usuario. Esto es útil para desarrollar proyectos completos, ya que Codex puede estructurar, planificar y crear múltiples archivos y carpetas de código interconectados.

Además, la API permite una comunicación iterativa, donde los desarrolladores pueden refinar sus solicitudes, hacer ajustes, realizar preguntas de seguimiento o pedir que el código sea revisado y mejorado. De esta manera, Codex actúa como un compañero de desarrollo, proporcionando sugerencias, corrigiendo errores y mejorando la calidad del

[14] https://platform.openai.com/docs/api-reference/making-requests

código generado en función de las especificaciones del usuario. Esto crea una experiencia de desarrollo mucho más ágil, reduciendo el tiempo dedicado a tareas repetitivas o complejas, y permitiendo a los programadores centrarse en la lógica de alto nivel o en las decisiones creativas del proyecto.

Create a human-like response to a prompt

```
import OpenAI from "openai";
const openai = new OpenAI();

const completion = await openai.chat.completions.create({
    model: "gpt-4o-mini",
    messages: [
        { role: "system", content: "You are a helpful assistant." },
        {
            role: "user",
            content: "Write a haiku about recursion in programming.",
        },
    ],
});

console.log(completion.choices[0].message);
```

Figura 9. Ejemplo de uso de la API extraído de la web de OpenAI

Como podemos observar en la Figura 9, la comunicación con la herramienta es muy sencilla, se llama a la API a través de la librería de manera muy intuitiva y se introduce el *Prompt* en el apartado de *content* dentro del segundo objeto de la lista *messages*, todo esto se encuentra perfectamente documentado en la web de la API oficial. Esto hace que no solo sea una herramienta extremadamente útil para los desarrolladores, sino que su uso es muy sencillo, lo que la convierte en una opción muy viable y muy recomendable, respaldado por el dominio de OpenAI en el mercado de los LLM.

Hay que recordar que estas herramientas, pese a todo el potencial que tienen, deben de ser usadas con cierta responsabilidad. Hemos visto que pueden usarse como un medio en el cual aprender muchos conocimientos y muy bien presentados de programación, pero igual programadores no expertos se pueden ver abrumados o pueden hacer un mal uso de estas herramientas, generando código por encima de sus posibilidades. Como se recalca en [10], que estudia el uso de OpenAI Codex, obteniendo resultados positivos de cara al futuro de la herramienta y recalcando su rápida evolución, pero también advirtiendo: “Having Codex in the hands of students should warrant concern similar to having a power tool in the hands of an amateur. The tool itself may not be intended to do harm, but with a vulnerable or untrained user, it may do just that.”

5.2. Github Copilot[15]

GitHub Copilot pertenece a la otra familia de modelos generadores que hemos visto, los asistentes de código. Está integrado directamente en entornos de desarrollo integrados (IDE) como Visual Studio Code y su función principal es sugerir líneas completas de código o incluso bloques enteros mientras el programador está trabajando. Copilot está impulsado por OpenAI Codex, presentado en el apartado anterior.

Copilot actúa como un copiloto en el proceso de desarrollo de software, ayudando con sugerencias contextuales y predicciones basadas en el código que el usuario ya ha escrito. Esto puede acelerar enormemente el trabajo, ayudar a evitar errores comunes y simplificar tareas repetitivas.

Origen

GitHub Copilot es el resultado de una colaboración entre GitHub y OpenAI. El proyecto se anunció en junio de 2021, con el objetivo de aprovechar la capacidad de OpenAI Codex para generar código y mejorar la productividad de los desarrolladores.

La idea detrás de Copilot es ofrecer una herramienta que funcione de manera intuitiva y natural dentro del flujo de trabajo del desarrollador. Al igual que un autocompletado avanzado, Copilot puede sugerir código en tiempo real, aprender del estilo y patrones del programador, y hacer recomendaciones relevantes basadas en el contexto del código actual.

Pre-Entrenamiento

Básicamente, al ser una herramienta basada en el modelo de Codex, el pre-entrenamiento de esta es el que ya hemos visto cuando hemos profundizado en el modelo desarrollado por OpenAI. Con la diferencia de que este entrenamiento le da la capacidad no solo de generar código que funcione, sino también de adaptarse al estilo de codificación del usuario.

Puede aprender de los patrones previos en el proyecto y generar sugerencias que se ajusten al contexto específico. Además, es capaz de entender descripciones en lenguaje natural, lo que le permite generar código basado en descripciones textuales.

¿Cómo se usa?

GitHub Copilot se utiliza como una extensión dentro de entornos de desarrollo como Visual Studio Code. Una vez instalada, la extensión sugiere automáticamente fragmentos de código a medida que el desarrollador escribe. Por ejemplo, si comienzas a escribir una función en Python, Copilot puede predecir y completar el cuerpo de la función basándose en su conocimiento del código anterior y del contexto.

Copilot funciona en varios lenguajes de programación, y es lo suficientemente flexible como para abordar tareas desde las más simples hasta las más complejas. Desde

[15] https://github.com/features/copilot

autocompletar líneas de código estándar, hasta generar funciones enteras o algoritmos complejos. Incluso puede ofrecer múltiples sugerencias para una sola tarea, y el usuario puede elegir la que mejor se ajuste a sus necesidades.

Los desarrolladores también pueden interactuar con Copilot de manera iterativa, refinar el código sugerido o hacer preguntas específicas en lenguaje natural, como "escribe una función para calcular la factorial de un número" y Copilot generará el código correspondiente, esto es una característica que suelen tener los asistentes de código que nos recuerda a los chatbots, y les añade mucho valor a esta familia de herramientas. Es una herramienta colaborativa que se adapta al flujo de trabajo del usuario, haciendo que la codificación sea más rápida y fluida.

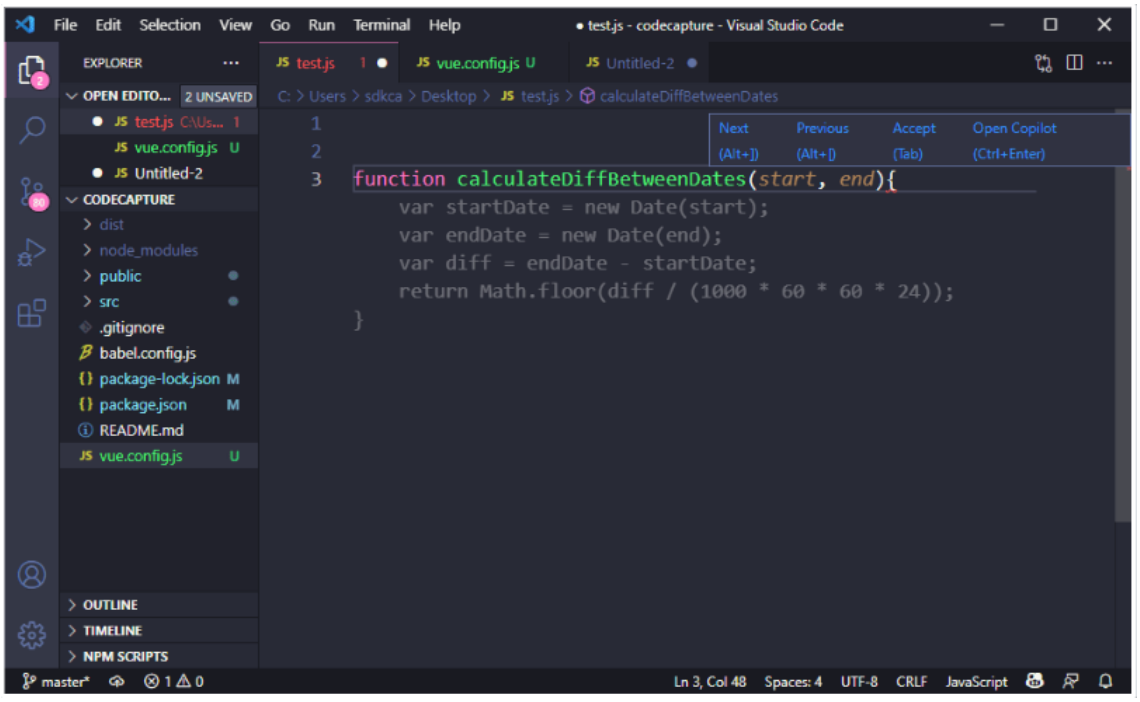


Figura 10. Ejemplo de Github Copilot en acción[16]

Como vemos en la Figura 10, el desarrollador escribe el nombre de la función, en este caso *calculateDiffBetweenDates*, con dos parámetros de entrada, *start* y *end*. Con esta información básica, GitHub Copilot es capaz de interpretar la intención del programador de manera intuitiva. Basándose en el contexto del nombre de la función y los parámetros, Copilot infiere que el objetivo es calcular la diferencia entre dos fechas.

A partir de esta suposición, la herramienta sugiere un método completo que incluye la lógica para calcular dicha diferencia, manejando correctamente los parámetros de entrada. Esta capacidad de Copilot para anticipar el propósito del código a partir de pistas contextuales, como los nombres de funciones y variables, permite que el desarrollador ahorre tiempo al no tener que escribir cada paso detalladamente.

[16] https://ourcodeworld.com/articles/read/1555/testing-github-copilot-how-good-is-it

Disponibilidad de la herramienta

GitHub Copilot no es una herramienta gratuita, que tiene sentido dado que Codex tampoco lo es, sin embargo, ofrece distintos planes a los usuarios que quieran emplearla. Existe un plan de suscripción estándar, de la misma manera que Codex, pero además GitHub ofrece GitHub Copilot de forma gratuita para estudiantes verificados y para contribuyentes activos a proyectos de código abierto. Estos grupos pueden beneficiarse del asistente de codificación sin pagar, lo que ayuda a fomentar el aprendizaje y el desarrollo dentro de la comunidad del software libre, lo que hace que la herramienta gane mucha popularidad y además Github hace hincapié en su compromiso con el futuro de la industria, dando facilidades a los desarrolladores que lo necesiten.

5.3. Code LlaMa[17]

Code LLaMA es un modelo de lenguaje especializado en la generación y comprensión de código de programación, desarrollado por Meta IA (anteriormente Facebook IA). Pertenece a la familia de modelos LLaMA, lanzados por Meta IA como parte de sus esfuerzos por liderar la investigación y desarrollo en inteligencia artificial y procesamiento de lenguaje natural. A medida que el campo de los modelos de lenguaje avanzaba, Meta IA reconoció la oportunidad de aplicar estas tecnologías al mundo de la programación, siguiendo el éxito de herramientas mencionadas anteriormente como GitHub Copilot y OpenAI Codex.

Específicamente, Code LLaMA está basado en LLaMA 2, que es la evolución de la primera versión de LLaMA y está diseñado para tareas generales de NLP, como generación de texto, traducción y análisis semántico. La principal característica de LLaMA 2 es su capacidad de generar texto coherente y contextual a partir de enormes volúmenes de datos textuales, utilizando una arquitectura basada en Transformers. Code LLaMA aprovecha la misma estructura y principios que LLaMA 2, pero su entrenamiento se enfoca específicamente en el código de programación, lo que le permite sobresalir en tareas relacionadas con la generación y comprensión de código en múltiples lenguajes.

LLaMA 2 es un modelo de código abierto y se lanzó en julio de 2023 como parte del compromiso de Meta IA de democratizar el acceso a modelos de inteligencia artificial de alto rendimiento. Fue entrenado con una amplia variedad de datos públicos, incluyendo libros, artículos, páginas web y otros textos en múltiples idiomas. Su diseño modular y escalable permite su uso en distintas aplicaciones, desde investigación hasta implementación comercial. Con su enfoque en mejorar la eficiencia y reducir los requisitos de hardware en comparación con otros modelos de lenguaje grande, LLaMA 2 ha logrado ser una herramienta poderosa para tareas de lenguaje natural, lo que también lo convierte en una base robusta para modelos especializados como Code LLaMA.

[17] https://github.com/meta-llama/codellama

Pre-Entrenamiento

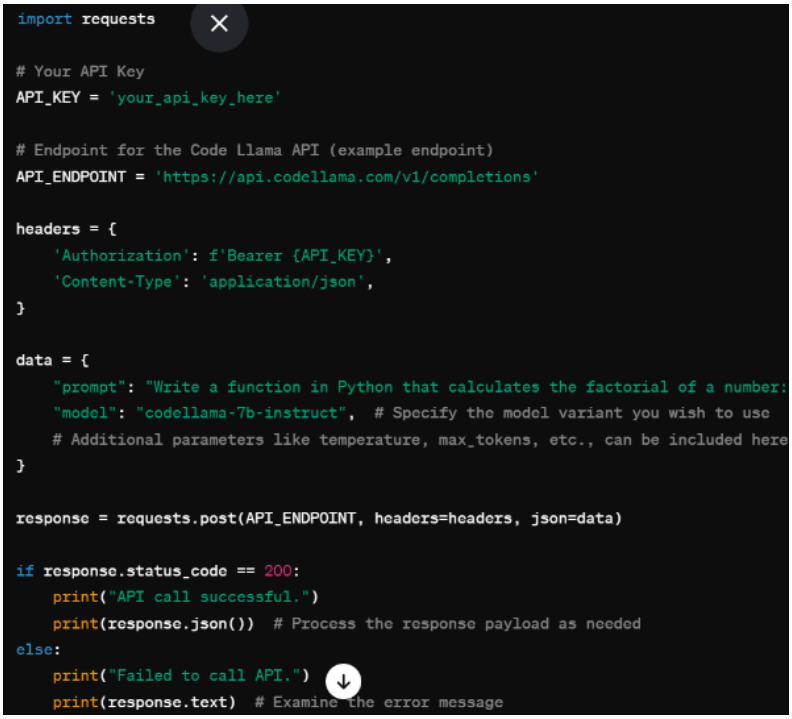
El entrenamiento de esta herramienta no difiere del resto, pese a que como ya hemos comentado, excepto pequeñas diferencias suelen ser muy parecido, en este caso Code Llama se basa en el uso de grandes conjuntos de datos que incluyen código fuente de lenguajes de programación populares como Python, JavaScript, Java, C++, y muchos más. Este conjunto de datos se complementa con documentación, libros de programación y ejemplos de código de diversas aplicaciones reales.

¿Cómo se usa?

A diferencia de las herramientas vistas anteriormente, que se empleaban o integradas en el entorno de desarrollo o mediante llamadas a una API, simulando un modelo conversacional, Code LLaMA junta estas dos maneras distintas de usar un LLM y puede ser empleada integrándola en el entorno de desarrollo o mediante su API.

Al igual que otras herramientas de codificación asistida por IA, los desarrolladores pueden interactuar con Code LLaMA mediante solicitudes en lenguaje natural o escribiendo fragmentos de código incompletos. Code LLaMA responderá completando el código, sugiriendo optimizaciones o corrigiendo errores.

Code LLaMA también admite interacciones iterativas, lo que significa que los usuarios pueden refinar las instrucciones y hacer ajustes sobre la marcha, obteniendo nuevas sugerencias que mejoran progresivamente la calidad del código. Esto es útil tanto para desarrolladores novatos como experimentados que desean acelerar su flujo de trabajo.



```python
import requests

# Your API Key
API_KEY = 'your_api_key_here'

# Endpoint for the Code Llama API (example endpoint)
API_ENDPOINT = 'https://api.codellama.com/v1/completions'

headers = {
    'Authorization': f'Bearer {API_KEY}',
    'Content-Type': 'application/json',
}

data = {
    "prompt": "Write a function in Python that calculates the factorial of a number:
    "model": "codellama-7b-instruct",  # Specify the model variant you wish to use
    # Additional parameters like temperature, max_tokens, etc., can be included here
}

response = requests.post(API_ENDPOINT, headers=headers, json=data)

if response.status_code == 200:
    print("API call successful.")
    print(response.json())  # Process the response payload as needed
else:
    print("Failed to call API.")
    print(response.text)  # Examine the error message
```

Figura 11. Ejemplo de uso de la API de Code LLaMA[18]

[18] https://meetrix.io/articles/how-to-install-code-llama-on-aws-via-pre-configured-ami/

En el ejemplo de la Figura 11, vemos cómo se implementa una comunicación sencilla con la API de LLaMA. Similar a la que hemos visto anteriormente con Codex, el programador crea un objeto con el Prompt que quiere enviar junto con el modelo, y recibe la respuesta con una simple invocación del método "requests.post". De igual manera que en Codex, y como tónica general para todas estas llamadas a la API (y para llamadas a APIs en general), la comunicación debe de ser todo lo fácil posible para que la API resulte llevadera de usar y atraer a más usuarios.

5.4. Amazon CodeWhisperer[19]

Amazon CodeWhisperer, como se puede intuir por su nombre, entra en el grupo de asistentes de código, capaz de autocompletar líneas de código, generar funciones enteras, y sugerir soluciones a problemas de programación, todo con el objetivo de mejorar la productividad y reducir los errores comunes.

Amazon CodeWhisperer fue lanzado como parte del ecosistema de *Amazon Web Services* (AWS), la plataforma de computación en la nube de Amazon que ofrece una amplia gama de servicios a nivel global. AWS es conocido por proporcionar infraestructura de tecnología de la información en la nube, desde almacenamiento y bases de datos hasta IA y *machine learning*. Dentro de este contexto, CodeWhisperer nace como una solución para mejorar la experiencia de los desarrolladores que utilizan AWS, ayudándolos a escribir código más eficiente y seguro para sus aplicaciones en la nube.

El desarrollo de CodeWhisperer es parte del compromiso de Amazon de facilitar la adopción de la nube y mejorar la productividad de los desarrolladores. Aprovechando su infraestructura y experiencia en machine learning, Amazon creó CodeWhisperer para integrarse perfectamente con otras herramientas y servicios de AWS, permitiendo a los desarrolladores construir, desplegar y gestionar aplicaciones en la nube con mayor rapidez y eficacia.

Pre-Entrenamiento

El pre-entrenamiento no difiere mucho de las técnicas que hemos visto anteriormente. Se basa en el análisis de una vasta colección de código fuente y documentación técnica disponible públicamente, así como en ejemplos de código específico del entorno AWS.

Una diferencia medianamente significativa que propone Amazon es que, al estar entrenado con datos relacionados con AWS, CodeWhisperer puede proporcionar sugerencias optimizadas para la nube, como la integración con servicios específicos de AWS, gestión de recursos en la nube y automatización de procesos.

[19] https://aws.amazon.com/es/q/developer/

¿Cómo se usa?

Como el resto de los asistentes de código, Amazon CodeWhisperer se utiliza principalmente como una extensión dentro de entornos de desarrollo integrados (IDE) como Visual Studio Code, JetBrains IDEs, y el propio AWS Cloud9 (recalcando lo que se ha dicho anteriormente sobre la integración de las diferentes herramientas de Amazon entre ellas). Una vez instalada la extensión, CodeWhisperer comienza a funcionar en segundo plano, observando el código que el desarrollador está escribiendo y ofreciendo sugerencias en tiempo real. Estas sugerencias pueden variar desde el autocompletado de una línea de código, hasta la generación de funciones completas o la recomendación de patrones de diseño.

El uso de CodeWhisperer es sencillo y se integra de manera natural en el flujo de trabajo del desarrollador. A medida que el programador escribe, CodeWhisperer analiza el contexto y propone fragmentos de código que podrían ser útiles. Los desarrolladores pueden aceptar, ignorar o modificar las sugerencias según sea necesario.

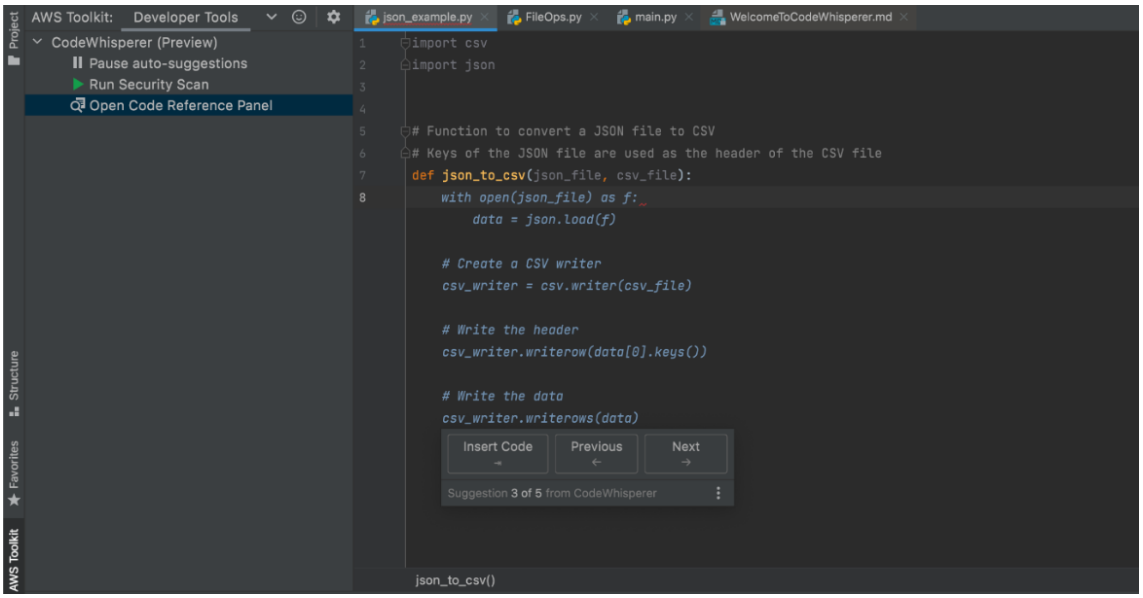


Figura 12. Ejemplo de uso de Amazon CodeWhisperer[20]

En la Figura 12 vemos un ejemplo propuesto en el blog de AWS, donde se propone un código básico para generar un fichero CSV, la herramienta genera el código en base a los comentarios de la función más el nombre y parámetros, como hemos visto de manera similar con otras herramientas de este estilo.

[20] https://aws.amazon.com/es/blogs/aws-spanish/presentamos-amazon-codewhisperer-el-companero-de-codificacion-basado-en-ml/

5.5. Replit Ghostwriter[21]

Replit Ghostwriter es una extensión avanzada desarrollada por Replit, una plataforma que permite a los desarrolladores escribir, ejecutar y colaborar en código directamente desde el navegador. Replit se ha convertido en un espacio integral para programadores de todos los niveles, albergando desde pequeños scripts hasta proyectos complejos en una variedad de lenguajes.

¿Qué es Replit?

Replit[22] es una plataforma en línea que ha revolucionado la manera en que los desarrolladores, desde principiantes hasta expertos, abordan la programación. A diferencia de los entornos de desarrollo tradicionales que requieren configuraciones locales complejas, Replit ofrece un entorno de desarrollo integrado (IDE) directamente en el navegador, eliminando la necesidad de instalaciones o configuraciones previas. Esto permite a los usuarios comenzar a codificar de inmediato, independientemente del dispositivo que estén utilizando.

Uno de los aspectos más atractivos de Replit es su soporte para una amplia gama de lenguajes de programación. Ya sea que estés trabajando en Python, JavaScript, C++, Ruby, HTML/CSS o incluso lenguajes menos comunes, Replit proporciona las herramientas necesarias para escribir, probar y ejecutar código en un entorno unificado y accesible. Además, su infraestructura en la nube asegura que los proyectos estén disponibles desde cualquier lugar, en cualquier momento, con solo iniciar sesión.

La plataforma también destaca por sus características colaborativas. Los desarrolladores pueden invitar a otros a trabajar en el mismo proyecto en tiempo real, similar a cómo se colabora en un documento de Google Docs. Esta función es especialmente valiosa para equipos distribuidos, estudiantes y profesores, ya que permite el trabajo conjunto sin las barreras tradicionales de las configuraciones de red o las versiones de software. También facilita el aprendizaje entre pares, ya que los usuarios pueden compartir código, hacer preguntas y recibir retroalimentación de la comunidad de Replit.

Replit no se limita a ser un simple IDE en línea. También ofrece una serie de herramientas y recursos educativos, como tutoriales interactivos, proyectos de código abierto y comunidades de codificadores, lo que lo convierte en un entorno ideal para aprender y enseñar programación. Además, su integración con GitHub y otras plataformas permite a los desarrolladores importar y exportar proyectos fácilmente, manteniendo un flujo de trabajo continuo entre diferentes herramientas.

[21] https://aws.amazon.com/es/q/developer/
[22] https://replit.com

Pre-Entrenamiento

Siguiendo la tónica del resto de herramientas vistas, el pre-entrenamiento de Replit Ghostwriter se basa en un vasto conjunto de datos que incluye código fuente de varios lenguajes de programación, ejemplos de proyectos reales, documentación técnica y otros recursos educativos. En este caso se añadirán al conjunto los proyectos y repositorios que puedan haber pasado por Replit, usando estos para alimentar a la herramienta.

¿Cómo se usa?

Como se ha visto con el resto de los asistentes, Replit Ghostwriter se utiliza directamente dentro del entorno de desarrollo en línea de Replit. Cuando los desarrolladores escriben código, Ghostwriter monitorea el proceso en tiempo real y ofrece sugerencias contextuales que pueden ayudar a completar fragmentos de código, corregir errores o incluso generar funciones completas basadas en la entrada del usuario.

El uso de Ghostwriter es intuitivo y está diseñado para integrarse sin problemas en el flujo de trabajo del desarrollador. A medida que se escribe código, Ghostwriter ofrece sugerencias que se pueden aceptar con un simple clic o mediante un atajo de teclado. Además, los desarrolladores pueden hacer preguntas en lenguaje natural, como "¿Cómo puedo crear una función para filtrar una lista en Python?", y Ghostwriter responderá generando el código correspondiente.

Al ser una herramienta desarrollada por el mismo organismo que el IDE donde opera, la integración es total y usar la herramienta es muy intuitivo para los usuarios. Cuando los desarrolladores escriben código en Replit, Ghostwriter monitorea el proceso en tiempo real y ofrece sugerencias contextuales que pueden ayudar a completar fragmentos de código, corregir errores o incluso generar funciones completas basadas en la entrada del usuario.

Que la herramienta esté integrada en el IDE, como lo hacían los anteriores asistentes que hemos visto, facilita mucho su uso, sin embargo, Ghostwriter lo lleva un escalón por encima, ya que como hemos dicho, el IDE en el que se usa es el desarrollado por Replit.

Coste de la herramienta

Replit ofrece un modelo de precios flexible, que permite a los usuarios elegir entre una versión gratuita y opciones de pago según sus necesidades. La versión gratuita de Replit proporciona acceso básico a la plataforma, incluyendo la capacidad de trabajar en proyectos personales o colaborar en tiempo real con otros desarrolladores. Sin embargo, la versión gratuita tiene algunas limitaciones, como una cantidad restringida de almacenamiento, memoria y tiempo de ejecución para los proyectos.

Para los usuarios que necesitan más recursos o características avanzadas, Replit ofrece planes de suscripción. Estos planes permiten acceso a más recursos de CPU, mayor capacidad de almacenamiento, ejecución de proyectos de mayor duración y otros beneficios como la colaboración en proyectos privados y soporte prioritario. Los precios de estos planes pueden variar según la cantidad de recursos necesarios y las características específicas que se deseen.

Replit Ghostwriter, por otro lado, es una característica premium dentro de la plataforma Replit y está disponible únicamente para los usuarios suscritos a uno de los planes pagos. Ghostwriter no está incluido en la versión gratuita de Replit. Para acceder a las capacidades de autocompletado, sugerencias de código, generación de funciones, y demás características de Ghostwriter, los usuarios deben suscribirse a un plan que incluya este servicio.

6. Herramientas generadoras de código en la práctica

Una vez visto tanto como funcionan de una manera abstracta como algunos ejemplos de herramientas, en este capítulo vamos a presentar resultados de estudios realizados sobre algunas de las herramientas vistas disponibles en la literatura, comprobando en base a unas métricas la calidad del producto final.

6.1. Revisión de estudios recientes en la literatura

Program Code Generation with Generative AIs [11]

En este trabajo los autores se centran en comparar la corrección, eficiencia y mantenibilidad del código de programas generados por humanos y por IA. Para ello, se analizaron los recursos computacionales de los códigos generados tanto por IA como por humanos, utilizando métricas como la complejidad temporal y espacial, así como el tiempo de ejecución y el uso de memoria. Además, se evaluó la mantenibilidad utilizando métricas como las líneas de código, la complejidad ciclomática, la complejidad de Halstead y el índice de mantenibilidad.

El estudio incluye generación de códigos en Java, Python y C++ por distintas IAs para resolver problemas de la plataforma de programación competitiva LeetCode[23]. Se seleccionaron seis problemas de diferentes niveles de dificultad, generando un total de 18 códigos por cada IA participante. Los resultados mostraron que GitHub Copilot fue la IA más efectiva, resolviendo el 50% de los problemas, mientras que otras IAs como CodeWhisperer no lograron resolver ninguno. Sorprendentemente, aunque ChatGPT solo generó cuatro códigos correctos, fue la única IA capaz de resolver un problema de alta dificultad.

En resumen, el estudio revela que, aunque las IAs tienen el potencial de ahorrar tiempo en la generación de código, solo un 20.6% de los códigos generados resolvieron correctamente los problemas planteados. Pese a la baja tasa de soluciones correctas, los resultados también sugieren que, para los códigos incorrectos, se pueden realizar modificaciones mínimas que permiten ahorrar entre un 8.9% y un 71.3% de tiempo comparado con programar desde cero.

[23] https://leetcode.com

	Easy			Medium			Hard						Total
	#1			#2			#3			#4			#5			#6			J|P|C|∑
	J	P	C	J	P	C	J	P	C	J	P	C	J	P	C	J	P	C	
ChatGPT (GPT-3.5)	—	—	✓	—	✓	✓	—	—	—	—	—	—	—	—	—	—	✓	—	0|2|2| 4
Bing AI (GPT-4.0)	—	✓	✓	✓	✓	✓	—	✓	✓	—	ϵ	—	—	ϵ	—	—	—	—	1|**3**|3| 7
GH Copilot (GPT-3.0)	✓	—	✓	✓	✓	✓	✓	—	✓	✓	—	✓	—	—	—	—	—	—	**4**|1|**4**| **9**
StarCoder	—	—	—	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1|0|0| 1
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	ϵ	—	—	—	—	—	0|0|0| 0
Code Llama (Llama 2)	—	—	—	✓	✓	✓	✓	—	—	—	—	—	—	ϵ	—	—	—	—	2|1|1| 4
InstructCodeT5+	—	—	—	—	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	0|1|0| 1
human	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	6|6|6|18

Figura 13. Tabla de resultados que muestra si el código generado es correcto

Este estudio refleja cómo de rápido avanzan estas tecnologías, como vemos en la Figura 13, donde dividen la tabla en columnas que representan tanto el nº del problema como el lenguaje en el que se resolvió, siendo Java, Python y C. Los resultados no son buenos, solo 1 de cada 5 generaciones aproximadamente fueron correctas, e incluso vemos algunos casos (señalados con un ϵ) donde el código generado no es ejecutable. Pese a esto, lo que se concluye que la ayuda de las herramientas en el ahorro de tiempo es significativamente notoria.

El trabajo no solo tiene en cuenta si el código es correcto o incorrecto, presenta una variedad de tablas donde se comparan número de líneas, complejidad ciclomática o tiempo de ejecución. Todo esto crea un *Maintainability Index*, que los autores usan para comparar la calidad en términos generales del código.

	Easy						Medium						Hard					
	#1			#2			#3			#4			#5			#6		
	J	P	C	J	P	C	J	P	C	J	P	C	J	P	C	J	P	C
ChatGPT (GPT-3.5)	—	—	53	—	**64**	60	—	—	—	—	—	—	—	—	—	—	**59**	—
Bing AI (GPT-4.0)	—	56	52	**59**	**64**	60	—	**64**	**60**	—	—	—	—	—	—	—	—	—
GH Copilot (GPT-3.0)	**52**	—	51	**59**	63	57	**59**	—	**60**	41	—	51	—	—	—	—	—	—
StarCoder	—	—	—	58	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Code Llama (Llama 2)	—	—	—	58	**64**	59	**59**	—	—	—	—	—	—	—	—	—	—	—
InstructCodeT5+	—	—	—	—	**64**	—	—	—	—	—	—	—	—	—	—	—	—	—
human	51	**62**	**57**	56	63	**62**	56	56	57	**59**	**60**	**59**	**33**	**39**	**34**	**50**	53	**48**
best AI vs. *human* (Δ in %)	**+2**	−10	−7	**+5**	**+2**	−3	**+5**	**+14**	**+5**	−30	—	−14	—	—	—	—	**+11**	—

Figura 14. Tabla de resultados que muestra el Maintainability Index del código generado

Vemos en la Figura 14 que el 50% de los códigos generados por las IAs tienen un mayor índice que el código generado por los humanos, dato que habla bien para los generadores, que, pese a que en el contexto del trabajo no han logrado buenos resultados en términos de código válido, el valor de este es igual o mejor que el generado por los programadores humanos.

Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation [1]

Como ya indicamos en el capítulo de introducción, los autores de este trabajo concluyeron que las herramientas analizadas requieren de un experto programador al lado suyo para que el código generado automáticamente pueda transformarse en algo de utilidad. Vamos ahora a profundizar en su propuesta para ver cómo realizaron el estudio y qué resultados obtuvieron.

El objetivo del trabajo es evaluar la calidad del código fuente generado por distintos motores de IA generativa, enfocándose en tres LLMs: GPT-3.5, GPT-4 y Google Bard. Seleccionaron tres problemas complejos de programación, el primero proponer un ejercicio de comparación de listas, el segundo, algo más complejo, donde tiene que generar un código que interactúe con el usuario y más adelante resuelva un problema simple de fracciones, y el ultimo, que pese a que requiere de realizar varias tareas, se basa mayormente en problemas de matrices, tocando así varios aspectos y poniendo a prueba un buen abanico de conocimientos dado que solo se trata de 3 problemas, y estos fueron resueltos por los distintos modelos, evaluando su funcionalidad y calidad mediante casos de prueba y métricas de Software.

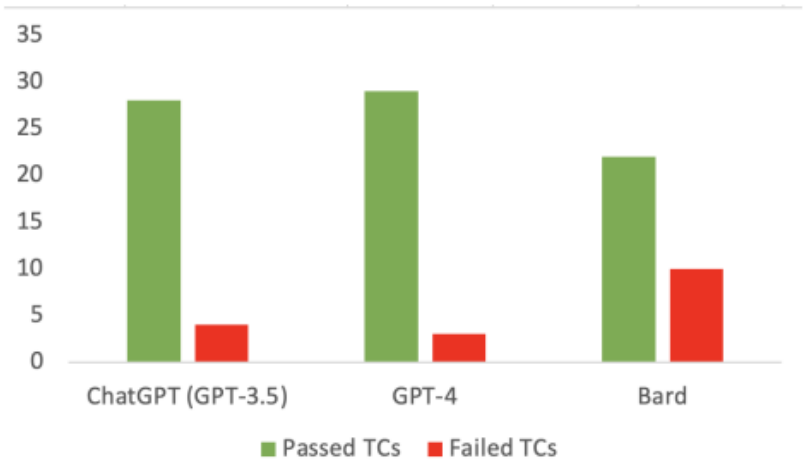


Figura 15. Resultados de las distintas herramientas probadas

Como vemos en la Figura 15, donde vemos representados la cantidad de casos de prueba (TCs) que ha pasado cada herramienta, pese a la conclusión a la que llegaron los autores del trabajo los resultados realmente son buenos, excepto Bard, que no logró resolver uno de los problemas propuestos, y su rendimiento en los otros fue peor que el de la familia de los GPTs.

De igual manera que el anterior estudio visto, y como suele ser una tónica general en los trabajos de esta naturaleza, la evaluación del código generado no es trivial, hay muchos factores que se tienen en cuenta para comprender realmente cómo de bien trabajan estas herramientas, como podemos ver en la Figura 16 a continuación.

	ChatGPT (GPT-3.5)	GPT-4	Bard
Lines of Code (LOC)	162	138	90
Number of Methods (NumOfMethod)	13	9	6
Cyclomatic Complexity	40	39	24
Cognitive Complexity	32	39	23
Bugs Detected	0	0	0
Vulnerabilities	0	0	0
Hotspots Reviewed	0	0	0
Code Smells	17	12	6
Duplications	0.00%	0.00%	0.00%

Figura 16. Tabla de parámetros del código generado en los 3 problemas

Los autores concluyeron que en general, los modelos LLM generaron código "*testeable*" para los problemas de programación presentados, aunque requirieron ajustes mínimos adicionales para corregir errores y asegurar su funcionalidad. Este proceso iterativo es el que hemos comentado anteriormente que resalta la necesidad de un programador humano que trabaje con las herramientas. Los modelos mostraron una alta tasa de éxito, superando la mayoría de los casos de prueba con pocas fallas, lo que refleja su robustez y utilidad en la generación de código. Aunque los problemas derivados de exámenes universitarios resultaron en soluciones concisas, estas presentaron cierta complejidad, sugiriendo la necesidad de refactorización y evaluación cuidadosa. A pesar de su complejidad moderada, el código no mostró errores, vulnerabilidades ni duplicaciones, demostrando alta calidad, fiabilidad y seguridad.

Este estudio, presentado en 2024, resalta los puntos que hemos tratado a lo largo del trabajo, de cómo estas herramientas, pese a poder presentar un alto conocimiento en técnicas avanzadas, manejo de librerías y la capacidad para resolver problemas de programación complejos, o su capacidad para generar código funcional y robusto en muchos casos, su producción no siempre cumple con los estándares de calidad necesarios para ser utilizados de manera confiable y sin supervisión.

Extending the Frontier of ChatGPT: Code Generation and Debugging [12]

La primera notable diferencia entre este estudio y el resto se ve claramente en el título, donde ya se indica que se trabajará con solo una herramienta, ChatGPT, más concretamente con la versión GPT-4. Si es cierto que, en momentos del trabajo, cuando GPT no es capaz de generar código que resuelva el problema, se apoya en herramientas como **LeetCode**[24], pero el modelo que genera el código principalmente es el desarrollado por OpenAI.

Solo nos queda hablar de los problemas seleccionados, que fueron extraídos de LeetCode y representan una variedad de dominios de programación como *Hash Table*, *Divide and Conquer*, *greedy*, y grafos, entre otros. Además, se consideraron problemas de

[24] https://leetcode.com

diferentes niveles de dificultad (fácil, medio y difícil) y se seleccionaron basándose en las tasas de aceptación de las soluciones en LeetCode para asegurar una representación equilibrada. La recopilación de datos se realizó antes de mayo de 2023 y se incluyó un total de 128 problemas, asegurando la diversidad tanto en los tipos de problemas como en sus niveles de complejidad.

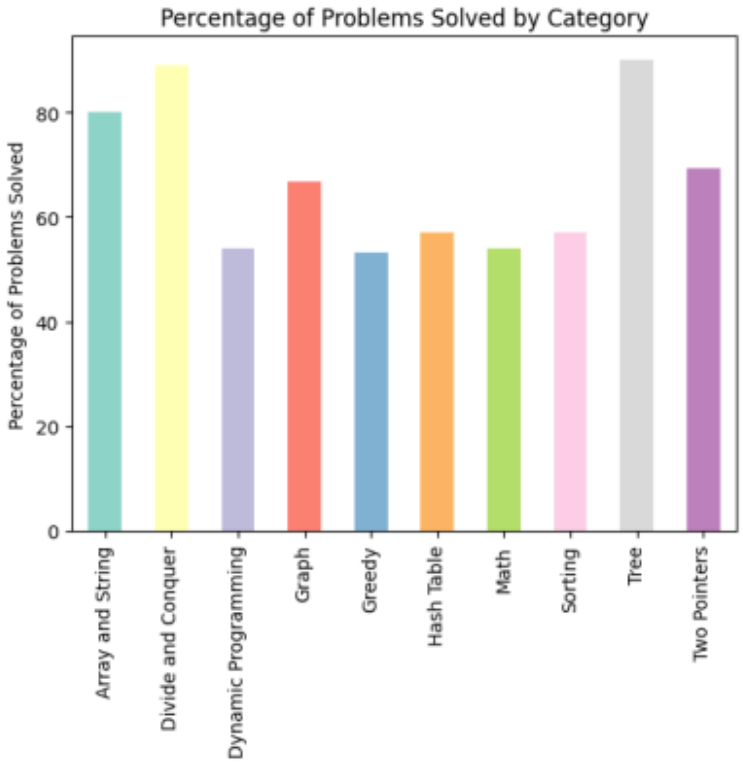


Figura 17. Porcentaje de problemas resueltos ordenados por categoría

Como observamos en la Figura 17, el modelo tuvo un rendimiento notable en problemas de dominios estructurados como *Tree* y *Divide and Conquer*, pero mostró dificultades con problemas más complejos, como los de *Dynamic Programing* y *Greedy*. Pese a que, como se ha puntuado reiteradas veces a lo largo del trabajo, estas herramientas, dada su naturaleza, van a tener un conocimiento técnico muy elevado, no siempre será sencillo plasmarlo en un código complejo.

En términos generales, ChatGPT mostró una tasa de éxito del 71.875%, resolviendo correctamente 92 de los 128 problemas del conjunto de datos. La mayoría de estos problemas fueron resueltos en el primer intento, mientras que para los problemas que había errado se pasaron a LeetCode, que se encargaba de generar sugerencias para arreglarlo, que posteriormente se le enviaban a ChatGPT, para ver si era capaz de rectificar y generar un código correcto.

Esta propuesta de trabajo es interesante, el dar al modelo una segunda oportunidad para ver si puede solventar los problemas puede abrir una puerta a modelos de trabajo iterativos que, en base a esta retroalimentación sean capaces de generar el código correcto

la mayoría de las veces. Sin embargo, los resultados no fueron buenos, como vemos en la Figura 18, los resultados obtenidos no fueron buenos, mostrando una clara cadencia en la herramienta a la hora de mejorar el código ya generado.

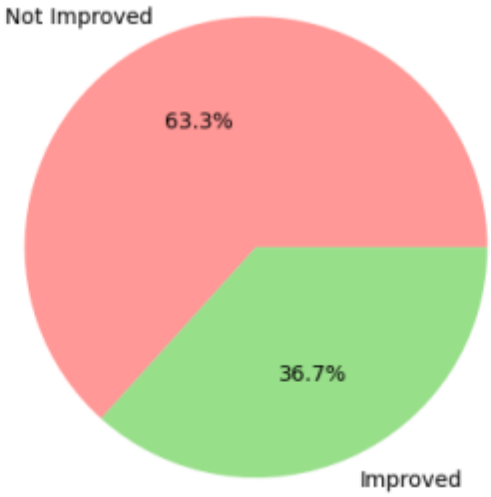


Figura 18. Representación de las veces que ChatGPT logró arreglar el código después del Feedback.

Como era de esperar, ChatGPT mostró mejores resultados en problemas más fáciles, con una tasa de éxito del 90%, mientras que su rendimiento disminuyó al 55% para problemas más difíciles, como se aprecia en la Figura 19. También se observó una correlación positiva entre la tasa de éxito del modelo y la tasa de aceptación de los problemas en Leetcode; tuvo más éxito en problemas con tasas de aceptación más altas.

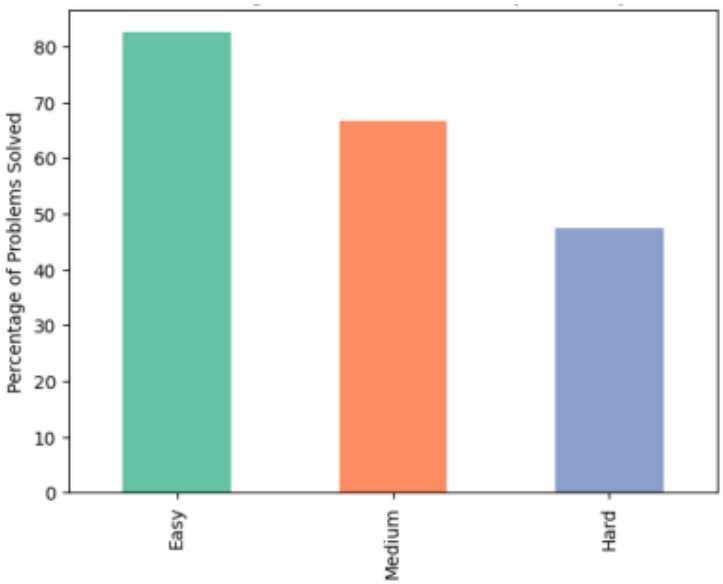


Figura 19. Distribución de problemas resueltos por dificultad.

Para concluir, los autores consideran que, aunque ChatGPT tiene un gran potencial para revolucionar la generación de código y asistir a los programadores, es necesario continuar mejorando y refinando sus capacidades, especialmente en lo que respecta a la incorporación efectiva de retroalimentación y el manejo de problemas más complejos.

Automatic Program Repair with OpenAI's Codex Evaluating QuixBugs [13]

Pese a que los estudios que hemos revisado tienen el mismo formato: elegir un conjunto de herramientas y problemas y ponerlas a prueba, las maneras de abordar este estilo de trabajo son muy amplias.

En este caso los autores se han centrado en *bugfixing*, investigando la capacidad del modelo Codex de OpenAI para reparar automáticamente errores en programas, utilizando el conjunto de datos QuixBugs [14], que es un conjunto de problemas de depuración que contiene implementaciones erróneas de 40 algoritmos clásicos en Python y Java.

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(a % b, b)
"""
Input:
    a: A nonnegative int
    b: A nonnegative int

Greatest Common Divisor

Precondition:
    isinstance(a, int) and
    isinstance(b, int)
Output:
    The greatest int that
    divides evenly into a and b
Example:
    >>> gcd(35, 21)
    7
"""
```

Figura 20. Ejemplo de un problema de QuixBugs en Python.

Para cada problema, como se puede observar en la Figura 20 el modelo recibe una breve descripción del error y el código incorrecto, y luego se le solicita generar una solución corregida. La evaluación se realizó midiendo cuántas veces Codex es capaz de reparar correctamente el error, comparando la salida generada por el programa corregido con la salida esperada. Se compara su rendimiento con técnicas existentes de reparación automática de programas como **PraPR**[25] y **GenProg**[26].

Los resultados muestran que Codex logró reparar correctamente el 50% de los errores de QuixBugs, superando a las herramientas tradicionales como PraPR y GenProg en la mayoría de los casos. Codex demostró ser particularmente eficaz en la reparación de errores sintácticos y de manipulación de estructuras de datos, donde se requiere un conocimiento profundo del lenguaje de programación y de la lógica del problema, característica que estos modelos suelen tener.

[25] https://github.com/prapr/prapr
[26] https://squareslab.github.io/genprog-code/

Sin embargo, también se identificaron limitaciones significativas. Codex falló en reparar algunos errores complejos, especialmente aquellos que requerían un razonamiento profundo sobre el flujo del programa o la integración de varios componentes del código. Además, el modelo mostró una tendencia a generar reparaciones que satisfacen casos de prueba específicos, pero no generalizan bien a casos adicionales, un fenómeno conocido como *overfitting* a los casos de prueba disponibles.

Testing LLaMA [15]

A lo largo del estudio, se evalúa el rendimiento de los modelos LLaMA en múltiples benchmarks, abarcando áreas como el razonamiento matemático, la generación de código, la comprensión de lectura y el razonamiento de sentido común, demostrando que los modelos LLaMA ofrecen resultados de alto nivel, comparables con modelos mucho más grandes y costosos de entrenar.

Pese a la amplitud de campos donde se estudia LLaMA, nos centraremos en las partes en las que estudia la generación de código, ya que son las que nos interesan para este trabajo. Aunque no sea un modelo específicamente entrenado para la generación de código, demostraron un alto rendimiento en estas tareas (evaluadas en los benchmarks *HumanEval* y MBPP[27]). Específicamente, el modelo LLaMA-65B superó al modelo PaLM-62B en ambos benchmarks. Esto es notable, ya que como se ha dicho, LLaMA no ha sido ajustado específicamente para la generación de código, pero aun así logra producir código de alta calidad. Los resultados fueron verificados utilizando métricas como pass@1 y pass@100, que indican la probabilidad de que el primer / uno de los 100 primeros intentos del modelo al generar código pase todas las pruebas unitarias propuestas por los BenchMarks.

6.2. Un ejemplo práctico de uso de LMMs para codificación

Vamos a realizar un estudio práctico a pequeña escala con un problema de programación relativamente simple y las herramientas que tengamos a nuestra disposición, usando las técnicas que hemos visto, aunque a una escala menor. Usaremos dos herramientas para nuestro caso práctico, GPT-4 y Microsoft Copilot, a las que hemos tenido acceso de forma gratuita, motivo por el que las hemos seleccionado.

El prompt

Como ya hemos mencionado varias veces a lo largo de la memoria, el Prompt es una parte fundamental para el uso efectivo de los LMMs, ya que éste debe aportar una descripción óptima del problema, ni muy extensa ni muy simple, y que contenga lo necesario para que la función resultante cumpla las características que deseamos. Veamos el caso de estudio que hemos preparado:

[27] https://paperswithcode.com/dataset/mbpp

"*Crea* un programa en Java que reciba una lista de enteros y un entero, el programa deberá de encontrar la sublista más larga dentro de la lista recibida cuya suma sea igual o menor que el número introducido. La cabecera será la siguiente:

public static List<Integer> sumaSubLista(List<Integer> lista, int target)"

Casos de prueba

Hemos diseñado un conjunto de casos de prueba donde examinamos las posibles situaciones a las que se puede enfrentar el programa, incluyendo casos con listas vacías, targets vacíos o números enormes. Para comprobar el correcto funcionamiento mostraremos por consola el resultado de la función y al lado el resultado esperado, usando este fragmento de código:

```
public static void main(String[] args) {
        System.out.println("Caso 1: " + sumaSubLista(List.of(), 5) + " -> []");
        System.out.println("Caso 2: " + sumaSubLista(List.of(5), 5) + " -> [5]");
        System.out.println("Caso 3: " + sumaSubLista(List.of(3), 5) + " -> [3]");
        System.out.println("Caso 4: " + sumaSubLista(List.of(7), 5) + " -> []");
        System.out.println("Caso 5: " + sumaSubLista(List.of(0, 0, 0, 0), 5) + " -> [0, 0, 0, 0]");
        System.out.println("Caso 6: " + sumaSubLista(List.of(2, 3, 5), 10) + " -> [2, 3, 5]");
        System.out.println("Caso 7: " + sumaSubLista(List.of(1, 2, 1, 2, 1, 2), 3) + " -> [1, 1, 1]");
        System.out.println("Caso 8: " + sumaSubLista(List.of(4, 5, 6, 7), 9) + " -> [4, 5]");
        System.out.println("Caso 9: " + sumaSubLista(List.of(4, 1, 2, 3, 5, 6), 6) + " -> [1, 2, 3]");
        System.out.println("Caso 10: " + sumaSubLista(List.of(1, -2, 3, -4, 5), 3) + " ->  [1, -2, 3, -4, 5]");
        System.out.println("Caso 11: " + sumaSubLista(List.of(0, 0, 1, 2, 3), 3) + " -> [0, 0, 1, 2]");
        System.out.println("Caso 12: " + sumaSubLista(List.of(-5), -5) + " -> [-5]");
        System.out.println("Caso 13: " + sumaSubLista(List.of(6, 7, 8), 5) + " -> []");
        System.out.println("Caso 14: " + sumaSubLista(List.of(1, 2, -1, -2, 0, 3), 0) + " -> [-2, -1, 0, 1, 2]");
        System.out.println("Caso 15: " + sumaSubLista(List.of(-1, -2, -3, -4, -5), -6) + " -> [-1, -2, -3, -4 , -5]");
        System.out.println("Caso 16: " + sumaSubLista(List.of(1000000), 5) + " -> []");
        System.out.println("Caso 17: " + sumaSubLista(List.of(0, 1, 0, 2, 0, 3, 0, -1, 0, -2, 0), 0) + " -> [0, 1, 0, 2, 0, 0, -1, 0, -2, 0]");
        System.out.println("Caso 18: " + sumaSubLista(List.of(100, -200, 300, -400, 500, -600, 700, -800), -100) +
                " -> [100, -200, 300, -400, 500, -600, 700, -800]");
        System.out.println("Caso 19: " + sumaSubLista(List.of(1, 2, 3, 4, 5), Integer.MAX_VALUE) + " -> [1, 2, 3, 4, 5]");
}
```

Figura 21. Código que imprime en consola el resultado de los casos de prueba propuestos

Algunos de los casos que pueden resultar más dudosos son aquellos que incluyen números negativos, ya que el método tiene que encontrar la sublista más larga cuya suma sea igual o menor que el *target*. En estos casos, añadir números negativos a la sublista puede ser siempre óptimo, ya que los números negativos permiten reducir la suma total de la sublista, lo que facilita cumplir con la restricción de ser menor o igual al *target*. Sin embargo, este enfoque puede ser más complicado cuando hay una combinación de números positivos y negativos, especialmente si el método no considera de manera adecuada cómo los negativos afectan la suma acumulada.

Resultados obtenidos usando la herramienta GPT-4

La primera herramienta que pondremos a prueba es ChatGPT, en concreto la versión que usa GPT-4, que, como ya se ha indicado, ofrece unas pruebas gratuitas diarias para usar el modelo. Además, como ya se ha visto en los trabajos que hemos revisado, la familia de los GPTs ha obtenido muy buenos resultados en comparación con otras herramientas basadas en LMMs. El código que ha generado GPT-4 a partir del prompt propuesto es el siguiente:

```java
public class main {
    public static List<Integer> sumaSubLista(List<Integer> lista, int target) {
        // Variables para manejar la ventana deslizante y almacenar resultados
        int inicio = 0, sumaActual = 0;
        int maxLen = 0, mejorInicio = 0, mejorFin = -1;

        // Iteramos por la lista para encontrar la sublista más larga
        for (int fin = 0; fin < lista.size(); fin++) {
            sumaActual += lista.get(fin);

            // Ajustamos la ventana deslizante si la suma excede el objetivo
            while (sumaActual > target && inicio <= fin) {
                sumaActual -= lista.get(inicio);
                inicio++;
            }

            // Comparamos la longitud de la sublista actual con la mejor encontrada
            if (fin - inicio + 1 > maxLen) {
                maxLen = fin - inicio + 1;
                mejorInicio = inicio;
                mejorFin = fin;
            }
        }

        // Si no se encuentra ninguna sublista válida, se retorna una lista vacía
        if (mejorFin == -1) {
            return new ArrayList<>();
        }

        // Retornamos la sublista más larga encontrada
        return lista.subList(mejorInicio, mejorFin + 1);
    }
```

Figura 22. Código generado por GPT-4

Por último, vamos a ver los resultados obtenidos por esta función. A continuación, se muestra la salida por consola que se ha obtenido al ejecutar el código generado por GPT-4 para los casos de prueba definidos:

```
Caso 1: [] -> []
Caso 2: [5] -> [5]
Caso 3: [3] -> [3]
Caso 4: [] -> []
Caso 5: [0, 0, 0, 0] -> [0, 0, 0, 0]
Caso 6: [2, 3, 5] -> [2, 3, 5]
Caso 7: [1, 2] -> [1, 1, 1]
Caso 8: [4, 5] -> [4, 5]
Caso 9: [1, 2, 3] -> [1, 2, 3]
Caso 10: [1, -2, 3, -4, 5] ->  [1, -2, 3, -4, 5]
Caso 11: [0, 0, 1, 2] -> [0, 0, 1, 2]
Caso 12: [-5] -> [-5]
Caso 13: [] -> []
Caso 14: [-1, -2, 0, 3] -> [-2, -1, 0, 1, 2]
Caso 15: [] -> [-1, -2, -3, -4 , -5]
Caso 16: [] -> []
Caso 17: [0, -1, 0, -2, 0] -> [0, 1, 0, 2, 0, 0, -1, 0, -2, 0]
Caso 18: [-200] -> [100, -200, 300, -400, 500, -600, 700, -800]
Caso 19: [1, 2, 3, 4, 5] -> [1, 2, 3, 4, 5]
```

Figura 23. Salida por consola del código generado por GPT-4

Podemos observar varias cosas interesantes en la Figura 23, en primer lugar, el desempeño del código generado no ha sido malo, 14/19 casos correctos, fallando los casos 7, 14, 15, 17 y 18, lo que representa aproximadamente que un 73'7% de los tests han sido superados en la primera iteración. Este resultado nos hace pensar que, como hemos visto en otros estudios y se ha recalcado a lo largo del trabajo, con la ayuda de un programador y algunos pequeños ajustes , el código puede ser completamente funcional.

Es interesante observar también en qué falla el código. Algo que tienen en común la mayoría de los tests para los que el código no ha dado el resultado correcto es que el *target* es un número negativo. Como hemos señalado anteriormente, es algo complejo tratar con estos números junto con positivos si no se hace correctamente. El programa no lee correctamente que la suma de muchos números negativos siempre va a ser menor que un número positivo, y muchas de las listas que propone para estos casos son incompletas.

Por ello, hemos iterado con la herramienta para que añada las modificaciones necesarias para que el código trabaje bien con los casos de números negativos, el prompt ha sido el siguiente:

"*Una vez ha sido testeado me he dado cuenta de que cuando se enfrenta a una lista de números negativos no lee correctamente que cuantos más sume más pequeña será, por lo que puede añadir a la sublista todos cuantos quiera. Arregla el código generado para que tenga esto en cuenta.*"

Y el código generado en esta segunda iteración es el que vemos en la Figura 24, donde vemos como la herramienta ha modificado la condición del if que pertenece al segundo bloque de código dentro del primer bucle for.

```java
public static List<Integer> sumaSubLista(List<Integer> lista, int target) {
    // Variables para manejar la ventana deslizante y almacenar resultados
    int inicio = 0, sumaActual = 0;
    int maxLen = 0, mejorInicio = 0, mejorFin = -1;

    // Iteramos por la lista para encontrar la sublista más larga
    for (int fin = 0; fin < lista.size(); fin++) {
        sumaActual += lista.get(fin);

        // Ajustamos la ventana deslizante si la suma excede el objetivo
        while (sumaActual > target && inicio <= fin) {
            sumaActual -= lista.get(inicio);
            inicio++;
        }

        // Solo se toma en cuenta la nueva sublista si su suma no excede el target
        if (sumaActual <= target && (fin - inicio + 1 > maxLen)) {
            maxLen = fin - inicio + 1;
            mejorInicio = inicio;
            mejorFin = fin;
        }
    }

    // Si no se encuentra ninguna sublista válida, se retorna una lista vacía
    if (mejorFin == -1) {
        return new ArrayList<>();
    }

    // Retornamos la sublista más larga encontrada
    return lista.subList(mejorInicio, mejorFin + 1);
}
```

Figura 24. Código generado por GPT-4 tras la iteración.

Pero veamos si realmente se han resuelto los casos que antes fallaba el código generado. La salida por consola tras la ejecución del nuevo código es la siguiente:

```
Caso 1: [] -> []
Caso 2: [5] -> [5]
Caso 3: [3] -> [3]
Caso 4: [] -> []
Caso 5: [0, 0, 0, 0] -> [0, 0, 0, 0]
Caso 6: [2, 3, 5] -> [2, 3, 5]
Caso 7: [1, 2] -> [1, 1, 1]
Caso 8: [4, 5] -> [4, 5]
Caso 9: [1, 2, 3] -> [1, 2, 3]
Caso 10: [1, -2, 3, -4, 5] ->  [1, -2, 3, -4, 5]
Caso 11: [0, 0, 1, 2] -> [0, 0, 1, 2]
Caso 12: [-5] -> [-5]
Caso 13: [] -> []
Caso 14: [-1, -2, 0, 3] -> [-2, -1, 0, 1, 2]
Caso 15: [] -> [-1, -2, -3, -4 , -5]
Caso 16: [] -> []
Caso 17: [0, -1, 0, -2, 0] -> [0, 1, 0, 2, 0, 0, -1, 0, -2, 0]
Caso 18: [-200] -> [100, -200, 300, -400, 500, -600, 700, -800]
Caso 19: [1, 2, 3, 4, 5] -> [1, 2, 3, 4, 5]
```

Figura 25. Salida generada por el nuevo código.

Como a simple vista se puede observar en la Figura 25, la salida por consola ha sido exactamente la misma que con el código generado antes de la iteración, lo que resalta la carencia que suelen tener estas herramientas (y que ya hemos comentado anteriormente) a la hora de examinar y modificar un código existente.

Resultados obtenidos usando la herramienta Copilot

La UPV, mediante la Plataforma ADIGITAL, más concretamente en el apartado de IA[28], ofrece una versión gratuita limitada de Copilot, no la versión de GitHub, sino una herramienta desarrollada por Microsoft, por lo que no está entrenada con código, pero al ser una herramienta que proporciona la UPV de manera gratuita, es interesante considerarla para este pequeño estudio. Al proporcionar a la herramienta el Prompt visto anteriormente, el código generado ha sido el siguiente:

[28] https://www.upv.es/contenidos/adigital/ia-2/

```java
public static List<Integer> sumaSubLista(List<Integer> lista, int target) {
    List<Integer> subLista = new ArrayList<>();
    int maxLength = 0;

    for (int i = 0; i < lista.size(); i++) {
        int sum = 0;
        List<Integer> tempSubLista = new ArrayList<>();

        for (int j = i; j < lista.size(); j++) {
            if (sum + lista.get(j) <= target) {
                sum += lista.get(j);
                tempSubLista.add(lista.get(j));
            } else {
                break;
            }
        }

        if (tempSubLista.size() > maxLength) {
            maxLength = tempSubLista.size();
            subLista = new ArrayList<>(tempSubLista);
        }
    }

    return subLista;
}
```

Figura 26. Código generado por la herramienta Copilot

Si lo introducimos en el entorno de desarrollo y ejecutamos, el resultado obtenido es exactamente el mismo que se obtuvo con GPT, no solo el mismo número de pruebas superadas, sino que en las pruebas donde ha fallado la sublista errónea obtenida ha sido la misma. No solo este hecho es curioso, sino que demuestra la gran capacidad y el gran potencial que tienen estas herramientas para generar código pese a no estar específicamente preparadas para ello, si bien todas ellas adolecen de las mismas carencias a la hora de ser usadas para la generación de código para resolver problemas de una cierta dificultad.

```
Caso 1: [] -> []
Caso 2: [5] -> [5]
Caso 3: [3] -> [3]
Caso 4: [] -> []
Caso 5: [0, 0, 0, 0] -> [0, 0, 0, 0]
Caso 6: [2, 3, 5] -> [2, 3, 5]
Caso 7: [1, 2] -> [1, 1, 1]
Caso 8: [4, 5] -> [4, 5]
Caso 9: [1, 2, 3] -> [1, 2, 3]
Caso 10: [1, -2, 3, -4, 5] ->  [1, -2, 3, -4, 5]
Caso 11: [0, 0, 1, 2] -> [0, 0, 1, 2]
Caso 12: [-5] -> [-5]
Caso 13: [] -> []
Caso 14: [-1, -2, 0, 3] -> [-2, -1, 0, 1, 2]
Caso 15: [] -> [-1, -2, -3, -4 , -5]
Caso 16: [] -> []
Caso 17: [0, -1, 0, -2, 0] -> [0, 1, 0, 2, 0, 0, -1, 0, -2, 0]
Caso 18: [-200] -> [100, -200, 300, -400, 500, -600, 700, -800]
Caso 19: [1, 2, 3, 4, 5] -> [1, 2, 3, 4, 5]
```

Figura 27. Resultados de la ejecución del código generado por Copilot

7. Conclusiones

En este trabajo hemos hablado de LLMs, de donde vienen o cómo funcionan, hemos visto que son la base de los generadores de código, y cómo estos últimos trabajan y se desenvuelven en estudios reales. Pero, ¿a dónde hemos llegado con todo esto?

En este trabajo, hemos visto cómo estas herramientas mejoran sus características a un ritmo muy alto, haciendo que sea difícil hacer predicciones precisas sobre su evolución futura. Sin embargo, lo que sabemos con certeza es que, en un futuro no muy lejano, serán utilizadas por la mayoría de los desarrolladores en una amplia variedad de escenarios. Estas herramientas no solo ayudarán a automatizar ciertas tareas, sino que también permitirán ahorrar una cantidad considerable de tiempo al abordar tareas repetitivas o complejas, facilitando así el flujo de trabajo y aumentando la eficiencia general en los proyectos de desarrollo de software.

No obstante, es importante destacar que, su viabilidad y efectividad aún dependen en gran medida de la intervención humana. Se ha recalcado constantemente que estas herramientas, para ser realmente útiles y consideradas viables en un entorno profesional, requieren de un desarrollador experimentado a su lado. Este desarrollador no solo revisa y mejora el código generado por la herramienta, sino que también se encarga de preparar Prompts de calidad, fundamentales para guiar a la herramienta de manera precisa y evitar resultados deficientes o irrelevantes.

Además, el rol del desarrollador es crucial para interpretar los resultados generados, entender sus limitaciones y hacer los ajustes necesarios para integrarlos adecuadamente en el proyecto en curso. A medida que estas herramientas avanzan, el enfoque no debería ser simplemente reemplazar al desarrollador, sino complementar sus habilidades, convirtiéndose en una extensión de su conocimiento y experiencia. De esta manera, el proceso de desarrollo no solo se vuelve más eficiente, sino que también permite a los desarrolladores centrarse en tareas más creativas y estratégicas, dejando que las herramientas se encarguen de los aspectos más rutinarios o repetitivos del trabajo.

Una pregunta recurrente, especialmente en sectores donde los avances tecnológicos dan lugar a la creación de herramientas o máquinas que realizan tareas de manera automática, es: ¿estas herramientas van a robarnos el trabajo? Esta preocupación es legítima y surge en casi cualquier campo que experimente un aumento en la automatización. En el caso del desarrollo de software, hemos observado que, por el momento, las herramientas de generación de código no son 100% fiables. Sin embargo, es casi seguro que su precisión y funcionalidad continuarán mejorando de forma constante hasta alcanzar niveles muy cercanos a la perfección, lo que podría permitirles generar código que sea completamente funcional y libre de errores en un futuro no muy lejano.

Pero el trabajo del desarrollador de software va mucho más allá de simplemente escribir líneas de código. Es un proceso altamente complejo que abarca múltiples fases, desde la concepción y el diseño de la arquitectura del software hasta la implementación, la prueba, la integración y el mantenimiento continuo del código. Aun cuando se logre generar código que funcione correctamente, existe un extenso conjunto de tareas que deben realizarse para asegurar que dicho código se ajuste al contexto específico para el cual se está desarrollando, que cumpla con todos los requisitos del proyecto, que sea escalable, seguro y mantenible, entre muchos otros factores.

En este punto, hemos mostrado en el trabajo que podemos descartar como candidatos a los modelos basados en chat conversacionales o generadores de texto plano. Estos modelos, aunque han demostrado ser útiles en ciertas tareas de codificación, tienen limitaciones significativas a la hora de generar código que se adapte de manera efectiva a contextos altamente específicos o complejos. A pesar de sus virtudes en otros aspectos, como la asistencia en la redacción de documentos o la generación de ideas, es prácticamente imposible que tengan en cuenta todos los matices y detalles necesarios para producir código altamente contextualizado y funcional en entornos reales de desarrollo.

Por otro lado, los asistentes de código integrados directamente en los entornos de desarrollo presentan una ventaja importante. Al estar integrados, tienen la capacidad de acceder al resto del código de la aplicación con la que se está trabajando, lo que les permite adaptarse mejor a las necesidades específicas del proyecto. Con los avances tecnológicos necesarios, estos asistentes podrían llegar a tener un conocimiento más profundo del contexto del software en desarrollo, permitiendo generar código que no solo sea funcional, sino también optimizado y alineado con los objetivos del proyecto.

Sin embargo, incluso con estos avances, la acción humana seguirá siendo indispensable. El desarrollo de software involucra múltiples facetas que requieren de juicio crítico, creatividad, toma de decisiones estratégicas y la capacidad de resolver problemas complejos en tiempo real. Estas son habilidades que, al menos por ahora, las máquinas no pueden replicar completamente (pero ya estamos hablando de avances de la tecnología en general, no de las herramientas estudiadas en este trabajo). Los desarrolladores humanos seguirán siendo necesarios para diseñar arquitecturas robustas, interpretar y validar los resultados generados por herramientas automatizadas, y asegurar que el producto final no solo funcione técnicamente, sino que también cumpla con los estándares de calidad y satisfacción del cliente.

En conclusión, aunque las herramientas de automatización y los asistentes de código seguirán evolucionando y desempeñando un papel cada vez más importante en el desarrollo de software, el papel del desarrollador humano no desaparecerá. Más bien, se transformará, permitiendo que estos profesionales se enfoquen en las tareas más estratégicas, creativas y de mayor valor añadido, mientras que las herramientas automatizadas se encargarán de las tareas más repetitivas o de soporte. Esta colaboración entre la tecnología y la habilidad humana es lo que realmente impulsará la innovación y el desarrollo en el futuro y nosotros, como desarrolladores, tenemos que estar dispuestos a aceptar este cambio y hacer que esta “relación” entre humano-máquina sea lo más fructífera posible.

8. Referencias

[1] Tosi, Davide. 2024. "Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation" Future Internet 16, no. 6: 188. https://doi.org/10.3390/fi16060188

[2] TURING, A. M. (1950), 'COMPUTING MACHINERY AND INTELLIGENCE', Mind LIX (236), 433-460.

[3] Weizenbaum, Joseph. "Computer power and human reason: From judgment to calculation." (1976).

[4] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is all you need." Advances in neural information processing systems 30.

[5] Chen, Banghao, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. (2023). "Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review." arXiv:2310.14735.

[6] Srivastava, A., BIG-bench authors. (2023). Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models. Transactions on Machine Learning Research. https://openreview.net/forum?id=uyTL5Bvosj

[7] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, Meng Yan. (2023). Improving ChatGPT Prompt for Code Generation. https://arxiv.org/pdf/2305.08360

[8] Yetiştiren, Burak & Özsoy, Işık & Ayerdem, Miray & Tüzün, Eray. (2023). Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. https://arxiv.org/abs/2304.10778

[9] Debalina Ghosh Paul, Hong Zhu and Ian Bayley. (2024). Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. https://arxiv.org/pdf/2406.12655

[10] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. (2022). The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In Proceedings of the 24th Australasian Computing Education Conference (ACE '22). Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[11] Idrisov, Baskhad, and Tim Schlippe. (2024). "Program Code Generation with Generative AIs" Algorithms 17, no. 2: 62. https://doi.org/10.3390/a17020062

[12] Fardin Ahsan Sakib, Saadat Hasan Khan, A. H. M. Rezaul Karim. (2023). “Extending the Frontier of ChatGPT: Code Generation and Debugging” https://arxiv.org/pdf/2307.08260

[13] Jonas A. Prenner, Roberto Minelli, and Andrea Mocci. (2021). Automatic Program Repair with OpenAI’s Codex: Evaluating QuixBugs. https://arxiv.org/pdf/2111.03922.

[14] Lin, Derrick, James Koppel, Angela Chen, and Armando Solar-Lezama. (2017). "QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge." In Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity, pp. 55-56.

[15] Touvron, Hugo & Lavril, Thibaut & Izacard, Gautier & Martinet, Xavier & Lachaux, Marie-Anne. (2023). LLaMA: Open and Efficient Foundation Language Models. https://arxiv.org/abs/2302.13971

9. Anexo

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No Procede**
ODS 1. **Fin de la pobreza.**				**X**
ODS 2. **Hambre cero.**				**X**
ODS 3. **Salud y bienestar.**				**X**
ODS 4. **Educación de calidad.**	**X**			
ODS 5. **Igualdad de género.**				**X**
ODS 6. **Agua limpia y saneamiento.**				**X**
ODS 7. **Energía asequible y no contaminante.**				**X**
ODS 8. **Trabajo decente y crecimiento económico.**	**X**			
ODS 9. **Industria, innovación e infraestructuras.**	**X**			
ODS 10. **Reducción de las desigualdades**.		**X**		
ODS 11. **Ciudades y comunidades sostenibles**.				**X**
ODS 12. **Producción y consumo responsables.**				**X**
ODS 13. **Acción por el clima.**				**X**
ODS 14. **Vida submarina.**				**X**
ODS 15. **Vida de ecosistemas terrestres.**				**X**
ODS 16. **Paz, justicia e instituciones sólidas.**				**X**
ODS 17. **Alianzas para lograr objetivos.**				**X**

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Cuando hablamos de avances tecnológicos, es difícil no pensar en cómo estos afectan todas las facetas de nuestra vida. En el caso de la informática, ocurre lo mismo: cualquier área descrita en los Objetivos de Desarrollo Sostenible puede verse impactada, de una manera u otra, por las mejoras en este sector.

Sin embargo, en la selección final, se ha intentado ser más conservador en cuanto a la influencia directa de las IAs generativas (el tema principal de este TFG) en algunos de estos ODS. A continuación, analizaremos cómo la adopción de estas tecnologías en el desarrollo de software podría tener un gran impacto en los ODS 4, 8, 9 y 10:

Educación de calidad:

Como hemos visto en el trabajo, un ejemplo muy claro de cómo estas herramientas mejoran la educación es cuando acompañan al código generado con una explicación, ayudando al programador a entender, e incluso mejorar sus conocimientos. No solamente se quedan en explicaciones de un código generado, sino que estas herramientas pueden funcionar como "compañeros", pudiendo preguntar cualquier cosa y obteniendo una respuesta en la mayoría de los casos correcta. Por lo tanto, podemos extrapolar esta característica para usar estas herramientas como "profesor", haciendo de ellas una fuente de educación viable no solamente para el campo de generación de código, sino para cualquier materia.

Esto podría ser especialmente beneficioso en contextos con escasos recursos, donde la personalización del aprendizaje puede ser fundamental para mejorar los resultados educativos y el tener acceso a una herramienta como esta puede catapultar el nivel y la cantidad de educación de calidad recibida por todo el mundo, no solamente en materia de programación, aunque sea el tópico del trabajo.

Trabajo decente y crecimiento económico:

La adopción de IAs generativas en el desarrollo de software puede contribuir significativamente al crecimiento económico mediante la automatización de tareas repetitivas y rutinarias, liberando así a los trabajadores para que se concentren en actividades que aporten un mayor valor añadido, como la resolución de problemas complejos, la creatividad, la innovación y la toma de decisiones estratégicas. Esta transformación puede incrementar la productividad y eficiencia en las empresas, al reducir costos operativos y mejorar la calidad de los productos y servicios ofrecidos. Por ejemplo, las IAs generativas pueden acelerar el desarrollo de software al automatizar partes del proceso de codificación, pruebas, depuración y documentación, lo que no solo reduce los tiempos de lanzamiento al mercado, sino que también mejora la precisión y disminuye los errores humanos.

Además, la creación de nuevos productos y servicios basados en inteligencia artificial puede abrir un abanico de oportunidades de empleo en sectores emergentes, como el desarrollo de modelos de IA, la implementación de sistemas automatizados, la ciberseguridad y el análisis de datos.

Industria, innovación e infraestructuras:

El enfoque de cara a la innovación va un poco de la mano con lo que hemos visto en el último punto, ya que al fin y al cabo la mayoría de las cosas descritas y las mejoras son innovaciones.

Además, pueden desempeñar un papel crucial en la modernización de la industria y el desarrollo de infraestructuras sostenibles. Estas tecnologías permiten optimizar procesos de producción automatizando la creación de software necesario para gestionar y mejorar operaciones industriales, lo que se traduce en mayor eficiencia y reducción de costos. Por ejemplo, los modelos generadores de código pueden crear soluciones

personalizadas para monitorear y controlar en tiempo real los sistemas de producción, minimizando los tiempos de inactividad y el desperdicio de recursos.

Además, estas IAs pueden mejorar el diseño de productos al generar automáticamente múltiples variantes de código para pruebas de calidad, lo que permite a las empresas desarrollar productos más innovadores y optimizados de manera más rápida. Esto es particularmente útil en sectores que requieren una rápida adaptación a las demandas del mercado, como el de la tecnología o la manufactura avanzada.

Reducción de las desigualdades:

El trabajo también guarda una cierta relación con el décimo ODS. La adopción de estas tecnologías puede incentivar el surgimiento de startups y fomentar el espíritu emprendedor, al permitir a pequeñas y medianas empresas competir en mercados dominados por grandes corporaciones gracias a soluciones más ágiles y personalizadas. Al mismo tiempo, las IAs generativas pueden apoyar a los emprendedores en la creación de prototipos y en la iteración de productos, reduciendo significativamente los costos de desarrollo iniciales, haciéndolas más asequibles a un sector empresarial más amplio que incluya pequeñas y medianas empresas, y acelerando el proceso de innovación en las mismas.