Evaluación de los LLMs para la generación de código

Carles Gallel Soler

Grado de Ingeniería Informática
Inteligencia artificial

Tutor/a de TF
Dr. Robert Clarisó Viladrosa
Profesor/a responsable de la asignatura
Dr. Xavier Baró Solé

20 de junio de 2023

Universitat Oberta de Catalunya

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Evaluación de los LLM para la generación de código*
Nombre del autor:	*Carles Gallel Soler*
Nombre del consultor/a:	*Dr. Robert Clarisó Viladrosa*
Nombre del PRA:	*Dr. Xavier Baró Solé*
Fecha de entrega (mm/aaaa):	*06/2023*
Titulación o programa:	*Grado de Ingeniería Informática*
Área del Trabajo Final:	*Inteligencia artificial*
Idioma del trabajo:	*Castellano*
Palabras clave:	*Inteligencia artificial, ChatGPT, Generación de código*

Resumen del Trabajo

La generación de código mediante LLMs (Large Language Models) es un campo que está en auge después de la popularidad de herramientas como ChatGPT o GitHub Copilot. Con ellas, la persona que desarrolla código mejora considerablemente el tiempo que invierte en escribir código fuente. No obstante, no se tiene la certeza que el código generado sea correcto.

Partiendo de este contexto, el objetivo de este trabajo es llevar a cabo un estudio que permita evaluar el grado de calidad que tiene el código generado aplicando distintas metodologías.

Para poder lograrlo, se ha definido una planificación con distintas tareas específicas. Concretamente, son:

1. Estudio del estado del arte donde se contextualice cuál es la situación actual del campo de la generación de código usando LLMs.
2. Diseño y desarrollo del prototipo que permita llevar a cabo todas las pruebas necesarias enviando consultas a la API del LLM.
3. Análisis de los resultados obtenidos, donde se lleve a cabo un estudio comparativo sobre todos los resultados obtenidos aplicando las distintas metodologías.

Como resultado, se conseguirán unas conclusiones que permitirán determinar

la viabilidad del uso de estas metodologías en la generación de código con LLMs y, además, permitirá determinar cuáles son las carencias actuales y las posibles mejoras que desarrollar de cara al futuro.

Abstract

Code generation using LLM is a burgeoning field following the popularity of tools such as ChatGPT or GitHub Copilot. With these tools, a programmer can significantly reduce the time it takes to develop code. However, there is no certainty that the generated code is correct.

In this context, the objective of this study is to conduct an assessment that allows the evaluation of the quality of the generated code when applying different methodologies.

To achieve this, a plan has been defined with several specific tasks. Specifically, these tasks are:

1. A study of the state of the art, contextualizing the current situation of the field of code generation using LLM.
2. The design and development of a prototype, which allows the execution of all the necessary tests by sending queries to the LLM API.
3. Analysis of the results obtained, conducting a comparative study on all the results obtained by applying the different methodologies.

As a result, some conclusions will be reached that will allow to determine of the viability of the use of these methodologies in code generation with LLM and, additionally, will allow the identification of current shortcomings and potential improvements to be developed in the future.

Índice

Lista de figuras

1. Introducción

En los últimos meses, los LLMs (*Large Language Model*) han experimentado un enorme crecimiento gracias a la popularización de herramientas como ChatGPT o GitHub Copilot.[1] No obstante, estas inteligencias artificiales ya tenían cierto recorrido dentro del campo de la computación. Concretamente, la primera versión pública del LLM que utiliza ChatGPT, GPT-1, fue publicada el año 2018 juntamente con un *paper* donde explicaban su funcionamiento.[2]

Continuando con el caso de ChatGPT, este ofrece múltiples opciones según lo que solicita la entrada que recibe: traducción, clasificación y generación de textos, buscador de información hasta la fecha que tiene constancia o, incluso, generación de código entre otros.[3] Es decir, la herramienta tiene la capacidad de generar código fuente según los requisitos que la persona solicita. Esto incluye el hecho de poder crear código en cualquier lenguaje y sobre cualquier tipo de algoritmo. Los LLMs son especialmente buenos en la generación de texto y, al final, el código fuente está formado por palabras con una sintaxis y una lógica concreta.

Sin embargo, según varios estudios[3], el código fuente generado no siempre mantiene una validez suficiente para poder entrar en producción de manera no supervisada y, además, el estilo de código utilizado por el LLM no siempre es el más adecuado. De hecho, debido a esta peculiaridad, ya ha habido propuestas donde se requieren múltiples versiones de un mismo algoritmo al LLM para que el usuario pueda elegir la que más se adapta a sus necesidades. En futuros apartados se hablará de ello con más amplitud.

En cualquier caso, queda claro cuál es la tendencia en cuanto al uso de estos LLM en la generación de código. Una vez se puedan conseguir herramientas deterministas, es decir, que ofrecen mucha más seguridad a la hora de generar el código, algunas de las tareas que actualmente se delegan a personas podrán ser delegadas directamente al LLM. Consecuentemente, es necesario llevar a cabo un estudio concluyente sobre el impacto que puede conllevar a escala social y profesional una herramienta como la que se acaba de proponer sin caer en sensacionalismos.

Así pues, el propósito de esta investigación es evaluar la generación de código utilizando LLMs y ofrecer unas conclusiones que permitan determinar la fiabilidad que tienen a la hora de generarlo y, además, poder determinar el grado de mejora del código generado según las diferentes metodologías que se apliquen.

Finalmente, con el objetivo de ofrecer brevedad y claridad en toda la memoria, se ha utilizado el lenguaje en masculino en algunos casos. Consecuentemente, cuando se haga referencia a una o varias personas de manera masculina, se refiere a cualquier persona independientemente de su género.

1.1. Contexto y justificación del Trabajo

Mejorar el rendimiento dentro de cualquier rama de la ingeniería siempre ha sido un objetivo perseguido por todos y la ingeniería informática no queda excluida. En los últimos años se han experimentado múltiples mejoras multidisciplinarias: mejora y optimizaciones de metodologías aplicadas, mejora de los entornos de desarrollo de software y, más recientemente, la incorporación de inteligencia artificial dentro de los IDE (*Integrated Development Environment*).

Como ya se mencionó en el apartado anterior, ya existen varias herramientas funcionales como GitHub Copilot o ChatGPT. Por lo tanto, la incorporación de estas en un entorno de trabajo mejoraría considerablemente la productividad. Sin embargo, es necesario poder garantizar que la herramienta utilizada sea determinista, ya que solo de esta manera se puede asegurar que el código generado es correcto. Así pues, existe una necesidad real de evaluar la calidad del código generado a partir de estos LLMs y, además, evaluar cuál es el grado de mejora aplicando ciertas metodologías. Concretamente, el estudio utiliza la metodología TDD (*Test Driven Development*) para asegurarse de que la calidad del código generado es la deseada.

Sin embargo, hay que tener en cuenta que la planificación se debe poder cumplir dentro del plazo de un semestre. En consecuencia, se ha decidido evaluar solo un LLM en lugar de hacerlo con varios, ya que la carga de trabajo hubiera hecho que no fuera factible llevar a cabo el trabajo. Además, en el momento de la planificación del trabajo, OpenAI solo tenía habilitada la API del LLM gpt-3.5-turbo porque desactivó la API de Codex y la API de GPT-4 solo es accesible a través de una lista de espera. Así pues, solo es viable llevar a cabo el estudio con el LLM gpt-3.5-turbo.

En cuanto al código que se quiere generar, teniendo en cuenta que el estado actual de los LLMs no permite enviar un contexto con muchos caracteres, el estudio se realizará con un tipo de código del cual tenga un mínimo de conocimiento. De esta manera, se ha decidido que se solicitará al LLM que genere código que represente ADT (*Abstract Data Type*) bastante utilizados: listas enlazadas, colas, pilas, algoritmos de ordenación, etc. Es decir, se solicitará que creen una clase de Java que ofrezca el comportamiento del ADT solicitado.

No obstante, dado que LLM probablemente habrá sido entrenado con bibliotecas públicas que ofrecen estos ADT, no es viable pedirle código como el de la biblioteca de Java. Consecuentemente, se debería disponer de una definición de estos ADT de la cual el LLM no tenga conocimiento. Adicionalmente, para poder valorar su validez y disponer de una colección de tests para enviar al LLM, se ha decidido utilizar la biblioteca de ADT de la asignatura de Diseño de Estructura de Datos de la UOC. En ella, hay un conjunto amplio de varios ADT de los cuales el LLM no tiene conocimiento.

Además, para no darle total libertad y obligarlo a implementar el ADT de una determinada manera, se le proporcionarán las cabeceras de los métodos que debe implementar. De esta manera, se provoca que no pueda proporcionar el mismo código con el cual ya ha sido entrenado. Es decir, tendrá que escribir código nuevo, aunque podrá tener un mínimo de conocimiento sobre lo que está generando. Asimismo, para saber qué debe hacer exactamente cada uno de los métodos solicitados, también recibirá el comentario asociado a él. Concretamente, dado que la biblioteca cuenta con comentarios en formado *javadoc*, recibirá esta información para entender cómo debe generar el código de ese método dentro de todo el contexto de la clase.

Es decir, el LLM recibirá un esqueleto con el nombre de la clase, la cabecera de los métodos que se quieren desarrollar juntamente con su comentario *javadoc*. La información que no recibirá en ningún momento serán los atributos necesarios para dar el comportamiento esperado al ADT. Tendrá que ser el propio LLM quien los declare según las necesidades que identifique.

Profundizando con el código que generará, también se ha decidido que el estudio se llevará a cabo **generando el código solicitado** de tres maneras diferentes: **sin validar con ningún tipo de test**, **validando el código con una colección de tests** que se debe aportar al momento de enviar la solicitud y, finalmente, **con una colección de tests que aportará el mismo LLM.**

De esta manera, se pretende analizar con qué metodología se ofrece una mejor respuesta. La calidad de esta dependerá del tiempo de ejecución que requiere el LLM para ofrecer el código, en caso de ser compilable, si es capaz de superar todos los tests, si su comportamiento es el esperado y si es lo más eficiente posible. Es decir, con qué metodología se ofrece un resultado que sea fiable y, en caso de que no se produzca este escenario con ninguna de las metodologías, que se pueda determinar con cuál se han obtenido mejores resultados.

El siguiente diagrama ilustra el funcionamiento de las tres metodologías gráficamente.

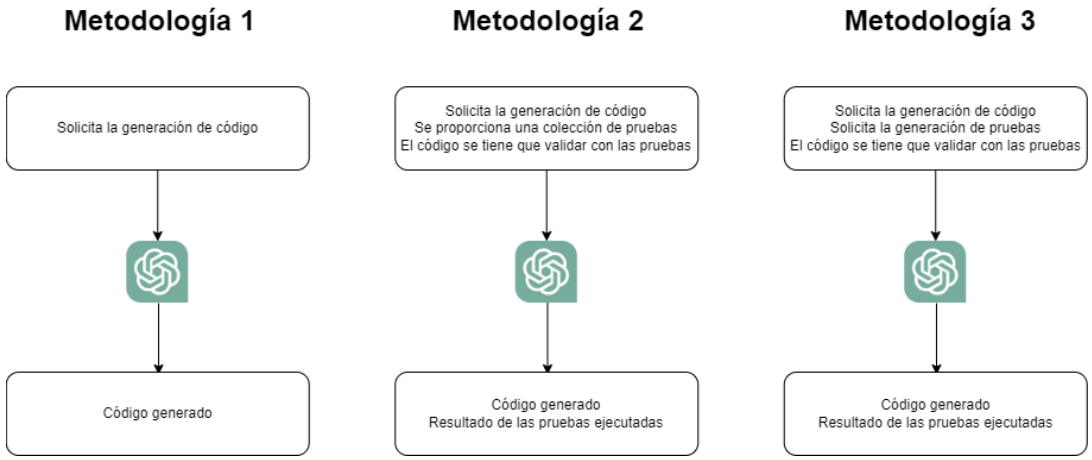


Figura 1. Diagrama que representa conceptualmente el funcionamiento de las tres metodologías aplicadas en este estudio. Fuente: original.

Asimismo, se ha decidido añadir otras metodologías que permitan mejorar la eficiencia temporal del LLM y la exactitud del código generado por este. De esta forma, el prototipo debe ofrecer múltiples formas de llamar a la API del LLM y analizar cuál de ellas es mejor. En concreto, por un lado, se intentará hacer las solicitudes con una única llamada a la API y, por el otro, se intentará utilizar una cadena con múltiples llamadas dividiendo el *prompt* y aportando ejemplos sobre lo que debe hacer.

Continuando con el punto anterior, también se ha decidido emplear *active prompting* dentro del prototipo para poder evaluar cuál es la mejora del código generado cuando se interactúa directamente con el LLM indicándole los cambios que debe hacer sobre el primer código fuente generado.

Adicionalmente, partiendo de la premisa de que en un futuro los LLM podrán ofrecer mejores resultados, aprovechando el prototipo que se ha desarrollado para llevar a cabo las pruebas, este trabajo también tiene como objetivo crear una extensión para Visual Studio Code donde se pueda enviar una solicitud a la API del LLM seleccionado y añada el código fuente generado directamente en el mismo IDE.

Finalmente, aprovechando la capacidad de poder usar *active prompting*, también se quiere dotar a la extensión de esta funcionalidad y que se puedan indicar las modificaciones necesarias dentro el mismo IDE.

1.2. Objetivos del Trabajo

Como se ha dicho en apartados anteriores, el objetivo principal de este trabajo es llevar a cabo una **evaluación sobre los LLMs en la generación de código** para analizar su fiabilidad. De esta manera, se podrán ofrecer unas conclusiones que determinen si es viable, o no, utilizar los LLMs como generadores de código y, además, poder dar una respuesta sobre cuál es la metodología que ofrece un mejor resultado.

Formalmente, este objetivo se descompone en los siguientes puntos:

- Evaluación de la calidad del código generado con diferentes metodologías.
- Evaluación de la eficiencia temporal según la metodología empleada.
- Evaluación de la mejora del código iterando con *active prompting*.

A continuación, también debe ofrecer una conclusión sobre si es posible generar código en entornos grandes. Es decir, hasta qué punto es capaz de generar el código en función del contexto que recibe.

En consecuencia, para poder garantizar el objetivo descrito, se deben considerar las siguientes tareas específicas.

- Estudio del estado del arte: como se ha dicho en la introducción, los LLMs ya tienen un cierto recorrido y, más concretamente, hay varios estudios donde ya tratan la generación de código utilizando LLMs. Por lo

tanto, se debe llevar a cabo un estudio sobre cuáles son las líneas de investigación que se están siguiendo en este campo, cuáles parecen que dan sus frutos y cuáles han sido ya descartadas porque son inviables. De esta manera, se logrará tener conocimiento sobre el estado actual del campo de estudio para poder profundizar sobre unas metodologías u otras.

- Diseño y desarrollo del prototipo: para poder realizar todas las pruebas necesarias, se debe diseñar y desarrollar un prototipo que permita hacer llamadas a la API de los LLMs estudiados aplicando todas las metodologías que se quieren analizar. Así, se logrará automatizar el proceso para poder obtener todos los resultados necesarios de cara al análisis de estos.
- Análisis de los resultados obtenidos: una vez se hayan lanzado todas las pruebas y se hayan recopilado todas las respuestas necesarias, se deben analizar desde diferentes variables: calidad del código generado, tiempo de espera durante la generación de código, etc. Después, el análisis debe permitir comparar los resultados entre metodologías y poder aportar unas conclusiones consistentes que permitan dar respuesta al estudio propuesto como objetivo principal.

En conclusión, el objetivo de este trabajo no pretende solucionar una problemática como tal, sino que busca evaluar si es viable, o no, y cuál es la mejor manera de hacerlo, el hecho de generar código a partir de lenguaje natural.

1.3. Impacto en sostenibilidad, ético-social y de diversidad

Las inteligencias artificiales están provocando cambios sustanciales de manera multidisciplinaria. Concretamente, tal como se ha visto en apartados anteriores, los LLMs pueden comportar cambios en disciplinas como la educación, la traducción de textos y, tal como se está observando a lo largo de este documento, en la ingeniería del software. Consecuentemente, se debe considerar el impacto que podría llegar a tener la herramienta que se está desarrollando en función de los resultados y de las conclusiones obtenidas.

Desde un punto de vista de sostenibilidad, se debe analizar teniendo en cuenta que esta herramienta mejoraría los costos temporales de los proyectos, ya que el hecho de poder generar el código utilizando un LLM acorta el tiempo de desarrollo considerablemente. Así, pues, aparecen dos impactos positivos: reducción de la energía empleada para desarrollar un proyecto y reducción de costos económicos.

El primero de los impactos citados está directamente relacionado con la programación verde, es decir, en la aplicación de metodologías que hagan que el desarrollo de un software sea respetuoso con el medio ambiente. En este caso lo sería por dos motivos: el código fuente se habría desarrollado en una cantidad de tiempo inferior al que se habría tardado desarrollando el código manualmente y, además, la herramienta tendría que haber usado el algoritmo más eficiente posible, hecho que permite usar la menor cantidad de recursos posibles durante su ejecución.

El segundo impacto positivo está relacionado con el ámbito económico. De nuevo, dado que el software se habría desarrollado con menos tiempo, los costos económicos disminuyen considerablemente, hecho que la convierte en una práctica muy atractiva para las empresas. Sin embargo, en una primera instancia se debería evaluar qué impacto tiene el uso de esta herramienta sobre el coste del software, ya que podría llegar a provocar bajadas de precio y hacer ciertos productos más accesibles económicamente hablando.

Adicionalmente, continuando con el punto del impacto en sostenibilidad, es importante hablar sobre la huella ecológica que ha tenido este proyecto. Teniendo en cuenta que un LLM es un modelo preentrenado, no se ha requerido entrenar ningún modelo para llevar a cabo el estudio. Ahora bien, dado que se han hecho múltiples consultas, sí que ha habido un consumo de energía.

En cualquier caso, el impacto es extremadamente inferior al que habría sido si se hubiera tenido que entrenar un modelo solo para el proyecto. Por lo tanto, se puede considerar que la huella ecológica es casi inexistente. Además, se ha tenido cuidado a la hora de elegir qué llamadas se hacían al LLM con el objetivo de minimizar ese impacto y obtener resultados más concluyentes.

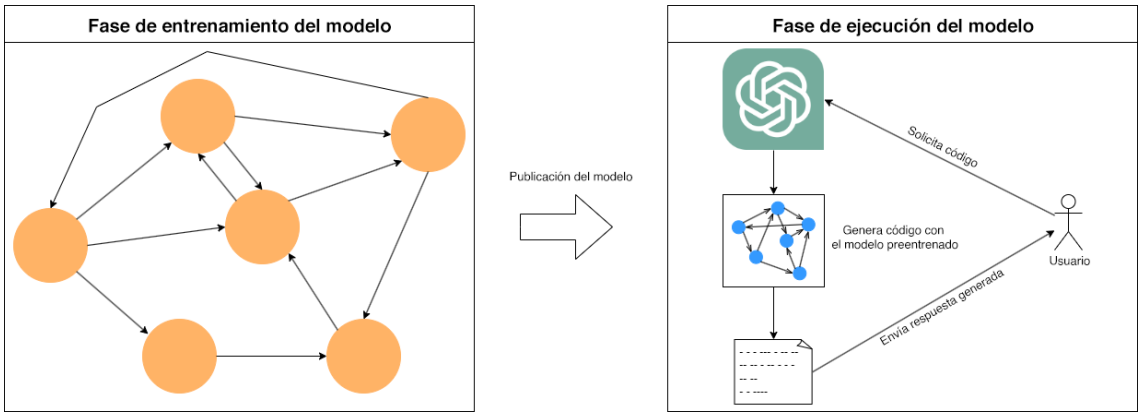


Figura 2. Diagrama conceptual que ilustra las dos fases por las que pasa el modelo. La fase de entrenamiento es la que provoca la huella ecológica. Fuente: original.

En un ámbito ético-social, el uso de esta herramienta, siempre considerando que se ha logrado que sea determinista, puede provocar que las empresas reduzcan el número de personal drásticamente, especialmente en aquellos puestos de trabajo donde su tarea sea generar un código fuente muy repetitivo. En cambio, si la herramienta a la que se llega no genera la suficiente confianza, el impacto social que tendrá será menor y su uso se limitará a acompañar a la persona que desarrolla el código para hacerla más eficiente. En cualquier caso, este escenario aparecería a medio o largo plazo, ya que debería pasar por una fase de asimilación donde las empresas sean capaces de incorporar esta herramienta en sus metodologías de trabajo. Buscar estas metodologías escapa del objetivo de este trabajo, pero es importante dejar constancia de ello.

Siguiendo dentro del mismo punto, también puede producirse un impacto positivo ya que puede acercar el desarrollo de software a personas que antes no tenían acceso. En consecuencia, sería más accesible y podría utilizarse

como puerta de entrada al campo del desarrollo de software. Sin embargo, es necesario destacar que para estas personas sería necesario tener un apoyo en todo momento para entender si se va por el buen camino: saber cuándo se deben modificar los resultados obtenidos y saber cómo se debe relacionar el código generado con el que ya existe. Dado que se está hablando de un caso en el que la persona no tiene conocimientos avanzados sobre lo que está haciendo, este punto se puede llegar a relacionar con el uso de los LLMs dentro de la educación, ya que el mismo LLM podría ser la figura que instruye y guía en todo momento durante el desarrollo de un código fuente.

A continuación, sin abandonar el impacto ético-social, se debe destacar que no se tiene conocimiento del conjunto de entrenamiento del LLM utilizado. Por lo tanto, existe la posibilidad de que parte de este conjunto de entrenamiento tenga licencias de propiedad intelectual. Hecho que puede comportar que el código generado por el mismo LLM contenga fragmentos de código protegidos intelectualmente y pueda acabar afectando en caso de incluirlo dentro de un producto.

En relación al impacto sobre la diversidad, relacionado con el punto anterior, una herramienta como la mencionada permitiría acercar el desarrollo de software a personas que no tienen acceso actualmente. Ya sea por una situación económica desfavorable que imposibilite acceder a este tipo de conocimientos o para romper la barrera que pueda haber debido a la complejidad del campo, con una herramienta como esta democratizaría el desarrollo de software.

Finalmente, es necesario dejar constancia de un último punto que afecta a cualquier proyecto donde haya una IA (inteligencia artificial) involucrada, que corresponde a la confidencialidad de los datos. En este caso, dado que no se está tratando con información confidencial, no existe ningún tipo de problema relacionado con la privacidad de datos.

1.4. Enfoque y método seguido

Para poder definir el enfoque y el método seguido durante el proyecto, se utilizará el ciclo de vida que define PMBOK[4]. Concretamente, un proyecto debe constar de cinco fases:

- Iniciación: se deben identificar las necesidades o problemas que se quieren cubrir o solucionar. Además, se debe analizar la viabilidad del proyecto. En el caso de este trabajo, esta fase se ha dado en la PAC 0.
- Planificación: se deben establecer y definir de manera clara cuáles son los hitos que cumplir junto con los requisitos para considerarlos completos y una estimación temporal dentro de un calendario. En este caso, se han llevado a cabo todos estos pasos en la PAC 1.
- Ejecución: es la fase en la que se deben completar todos los hitos que se han planificado. Suele ser la fase más larga de todas. De hecho, así ha sido en este trabajo y corresponde a todas las PAC desde la número 2 hasta la última, exceptuando la defensa del trabajo. En ellas, se han

ido completando los hitos planificados, incluyendo la redacción de la memoria y la presentación del proyecto.

- Seguimiento y control: es la fase en la que se verifica que todos los hitos se están cumpliendo según la planificación y, en caso de ser necesario, aplicar acciones de mitigación. Se puede considerar una fase que está presente durante todo el proyecto. Sin embargo, las tareas de seguimiento y control tuvieron lugar al final de las fases a partir de la PAC 2 hasta el final, donde se tuvo que actualizar el diagrama de Gantt que se desarrolló en la fase de planificación.
- Cierre: es la última fase del proyecto y es donde se deben incluir todas las tareas para poder finalizar el proyecto adecuadamente. Esta fase comparte otras PAC que se han clasificado como fase de ejecución. Concretamente, se está hablando de la memoria, la presentación y la defensa del proyecto. Aunque es cierto que corresponden a la parte de ejecución del trabajo, también se pueden considerar parte de la fase de cierre, ya que son la fase final del trabajo y permiten finalizar el proyecto, concluyendo con la defensa de este.

Consecuentemente, todo el proyecto ha seguido una metodología en cascada teniendo en cuenta que durante la fase de ejecución se han tenido que hacer ligeras modificaciones según los primeros resultados que se habían obtenido.

Finalmente, dejando constancia del trabajo que se llevó a cabo en la PAC 1, se deben mencionar posibles riesgos que se identificaron inicialmente. Formalmente, los riesgos identificables al inicio del proyecto han sido:

- Problemas técnicos a la hora de desarrollar el prototipo. Existe la posibilidad que aparezcan dificultades técnicas durante el desarrollo del prototipo. Como acción de mitigación se planificó utilizar días adicionales para algunas tareas, ya que algunas de estas se planearon con varios días para dejar margen de maniobra.
- Falta de conocimiento de las herramientas actuales. El desarrollo del prototipo requiere una investigación que permita determinar qué herramientas y bibliotecas son las más adecuadas para crearlo. Esto puede comportar un sesgo temporal dentro de la planificación si se requiere más tiempo del planificado. Como acción de mitigación se decidió emplear parte de los días adicionales de algunas de las tareas más simples.
- Los resultados no son los esperados. Dado que el trabajo está orientado a llevar a cabo un estudio sobre la generación de código, existe el riesgo de que los resultados no sean los esperados. Para mitigar este riesgo, se determinó que se deberían priorizar los objetivos y no podría desarrollar el prototipo para Visual Studio Code. En cualquier caso, los resultados deberían ser analizados y documentados en la memoria, determinando que la investigación no ha ido como se esperaba inicialmente.
- Cambio en la política de servicios. El estudio usa herramientas controladas por terceros, es decir, se depende en todo momento del mantenimiento del servicio por parte de una empresa ajena al proyecto. Consecuentemente, existe un alto riesgo de que el servicio cierre

teniendo en cuenta los antecedentes que se han dado, como es el caso de Italia. Como acción de mitigación, se determinó que se utilizaría una VPN (*Virtual Private Network*) para poder hacer llamadas a la API, en caso de que el cierre sea geopolítico. Si el cierre es global, el proyecto no se podrá llevar a cabo y no hay acción de mitigación posible.

- Resultados cambiantes según la versión. El LLM al cual se hacen las llamadas, GPT, tiene varias actualizaciones cada cierto tiempo. Estos cambios pueden provocar que los resultados obtenidos sean diferentes según la versión. Como acción de mitigación, se deberá informar de este suceso y, además, se deberán usar los días adicionales para intentar reconducir el resultado con la nueva versión.

1.5. Planificación del Trabajo

Para poder llevar a cabo la planificación de este trabajo, se ha tenido en cuenta que se debe poder completar dentro de los plazos de un semestre del curso académico universitario. Es decir, hay unas fechas clave marcadas por el plan docente de la asignatura. Consecuentemente, se han dividido las tareas agrupadas por las entregas parciales asignadas dentro del aula.

Adicionalmente, se ha considerado que la dedicación sería a tiempo parcial de lunes a sábado. Por lo tanto, las tareas que se han planificado contemplan un total de 300 horas de trabajo desde el inicio del estudio hasta la defensa de este. Así pues, la planificación inicial se puede consultar en el Anexo I de este documento.

Como se puede ver en el anexo, la planificación está organizada por entregas parciales donde, de manera resumida, se explicará cuáles son las tareas que se han planificado.

- PAC 0: es la fase en la cual se ha contextualizado el estudio y se ha llevado a cabo una investigación sobre cuál es el estado actual del campo de estudio propuesto.
- PAC 1: fase en la que se ha planificado todo el trabajo. Dentro de esta fase se ha considerado dar continuidad al proyecto más allá de la entrega final con la redacción de un *paper* si los resultados que se obtienen se consideran suficientes para ser publicables.
- PAC 2: se ha organizado de manera que sea la primera fase de trabajo donde se desarrolla la base del prototipo para poder llevar a cabo las pruebas y donde se empiezan a evaluar los primeros resultados para determinar la continuidad, o no, de la planificación inicial. Es decir, si se detecta que se debe realizar algún cambio, debe hacerse en este punto. Además, es la fase donde se analizan los resultados obtenidos aplicando las diversas metodologías TDD en la generación de código.
- PAC 3: es la segunda fase del trabajo donde se da continuidad a todo el trabajo realizado en la fase anterior teniendo en cuenta las posibles modificaciones y, además, se añaden nuevas metodologías para ser estudiadas. Asimismo, es donde se llevan a cabo todas las pruebas restantes para poder analizar los resultados.

- PAC 4: es la fase donde se ha planificado la redacción de la memoria y se finalizan los objetivos replanificados en caso de que los haya.
- PAC 5a: fase en la que se preparará una presentación del trabajo con material audiovisual.
- PAC 5b: se ha planificado responder a las preguntas del tribunal.
- Redacción del *paper*: es una fase fuera de plazo del semestre universitario donde, según los resultados obtenidos, se redactará un *paper*.

Ahora bien, dado que una de las tareas se consideró opcional en todo momento, la de desarrollar una extensión para Visual Studio Code, y no hubo tiempo para realizarla dentro de la planificación inicial, se llevan a cabo acciones de mitigación con el objetivo de replanificarla como una tarea de la siguiente entrega.

Asimismo, después de observar los primeros resultados, se tuvieron que modificar algunas de las tareas que se habían planificado porque se consideró que no tenía mucho sentido explorar algunas vías. Concretamente, se está hablando de la tarea donde se quería aplicar la metodología *chain-of-thought* para mejorar la calidad del código generado. En futuras secciones se argumentará el porqué de esta decisión con más detalle y se relacionará con la generación de código creando tests propios.

Así pues, la planificación ha tenido que ser modificada durante el transcurso del proyecto, aplicando acciones de mitigación donde ha sido necesario. La tabla de metas con la planificación final se puede consultar en el Anexo II.

Finalmente, para formalizar toda esta planificación y presentarla adecuadamente, se ha creado un diagrama de Gantt donde están listadas cada una de las metas que se han llevado a cabo durante el trabajo. Junto con cada meta, se ha descrito la condición necesaria para considerarla completada. Para verlo en detalle, se puede consultar el Anexo III.

1.6. Breve sumario de los productos obtenidos

Aunque se trata de un trabajo donde el objetivo principal no es crear o desarrollar un producto final, como se ha mencionado en el apartado de los objetivos, sí se busca crear una herramienta que sea capaz de generar el código utilizando la API de un LLM. Sin embargo, para poder llevar a cabo el estudio es necesario contar con un prototipo que permita hacer llamadas de manera automática y poder generar el código en función de una entrada de texto.

Consecuentemente, al final de este trabajo se obtendrá un **prototipo** funcional con el cual se podrá generar código mediante la API de un LLM y, además, sea capaz de interactuar con la persona solicitante usando *active prompting*. El código fuente del prototipo se puede consultar en el Anexo IV.

De manera paralela, también se obtendrán una **colección de gráficos** que permitirán entender la evaluación que se está llevando a cabo. Es decir, a partir

del código generado, se analizarán los resultados y se sintetizarán en diferentes gráficos que se usarán a lo largo de la memoria para complementar de manera gráfica los resultados alcanzados y las conclusiones.

Finalmente, a partir de gran parte del código fuente del prototipo, también se obtendrá una **extensión para el IDE Visual Studio Code** que será capaz de añadir el código generado, con la metodología que ofrece el mejor resultado, dentro del archivo de código fuente en el que se esté trabajando según una entrada que proporciona un usuario en lenguaje natural.

1.7. Breve descripción de los demás capítulos de la memoria

Justo después de la introducción de este trabajo, en el capítulo 2, se contextualiza cuál es el estado del arte sobre los LLMs y, más concretamente, sobre la generación de código utilizando los LLMs. Adicionalmente, se describe cuál es el estado actual de las principales herramientas que usan LLMs. A continuación, se hace una especial mención a cuáles son las metodologías que se están aplicando en la generación de código mediante un LLM. Para finalizar, se habla sobre el estado de los IDE que están incorporando generadores de código.

Es decir, con el capítulo 2, se pretende dar un contexto sobre la situación actual al lector de esta memoria. Además, toda la información que se encuentra en él corresponde a la síntesis de la investigación que se ha llevado a cabo durante la primera fase del proyecto, donde se ha tenido que contextualizar el campo de estudio para tener un punto de partida.

Seguidamente, en el capítulo 3, se expone el diseño del prototipo utilizado durante el estudio. Específicamente, se menciona la plataforma de ejecución para la que está pensado, se argumenta el porqué del lenguaje usado, se explica cuál es su funcionamiento y cómo se ha diseñado conceptualmente antes de desarrollarlo. Adicionalmente, se muestran imágenes que ayudan a entender visualmente la arquitectura que sigue el código del prototipo.

A continuación, el siguiente capítulo, el 4, trata de cómo se ha desarrollado el código para todos los requisitos y funcionalidades que debe ofrecer el prototipo. Además, se argumenta el porqué de las decisiones tomadas en función de los primeros resultados que se obtuvieron.

En cuanto al capítulo 5, es donde se lleva a cabo un análisis comparativo entre todos los resultados obtenidos de manera sintetizada y tratada. Es decir, una vez lanzadas todas las pruebas a la API del LLM, se ha realizado un análisis exhaustivo para analizar cada una de las respuestas ofrecidas para poder obtener las primeras conclusiones. Es la sección donde se representan los resultados de manera gráfica.

Una vez analizados los resultados, el capítulo 6 detalla la implementación de la extensión de Visual Studio Code utilizando la metodología que haya ofrecido mejores resultados.

Para finalizar, los capítulos 7 y 8 ofrecen los resultados finales de todo el estudio y las conclusiones que se pueden extraer de este. Cabe destacar que en esta sección es donde se argumenta si se dará continuidad, o no, al estudio con la redacción de un *paper*.

2. Estado del arte

Durante los últimos años, los LLMs han evolucionado a un ritmo vertiginoso y, actualmente, la tendencia sigue siendo la misma. Por lo tanto, el estado en el que se encontraban los LLMs ha ido variando durante el transcurso de este estudio. Consecuentemente, con el objetivo de ofrecer la mejor contextualización posible para el lector de esta memoria, la búsqueda de información que se ha llevado a cabo en este capítulo se ha intentado actualizar hasta el último momento antes de entregar este documento. Es decir, se han incluido las últimas actualizaciones hasta el día 20 de junio de 2023.

2.1. Situación actual de los LLMs

Gracias a los enormes avances que ha habido en los últimos años dentro de los LLM, cada vez se están obteniendo mejores resultados. Asimismo, los conjuntos de entrenamientos utilizados son más y más grandes, lo que dota a los LLMs con la capacidad de poder ofrecer respuestas más precisas según la entrada recibida.[3] Además, cada vez surgen nuevas herramientas que, aprovechando el potencial de los LLMs, permiten mejorar los resultados conseguidos. Es decir, se está empezando a crear una tendencia en la que se utiliza un LLM como parte central de un modelo, teniendo como soporte otras herramientas que le permiten mejorar el resultado.

Ahora bien, múltiples investigaciones advierten que existen algunas limitaciones, como la limitación de *tokens* que existe al enviar una entrada o la falta de memoria que tienen a largo plazo.

Debido al gran auge de un LLM en concreto, ChatGPT, gran parte de los *papers* de los últimos meses utilizan este modelo para llevar a cabo sus investigaciones y pruebas. De hecho, en una publicación donde se hace una síntesis sobre 194 *papers* en los que se analizan o se proponen nuevos modelos o metodologías, la palabra más mencionada es ChatGPT.

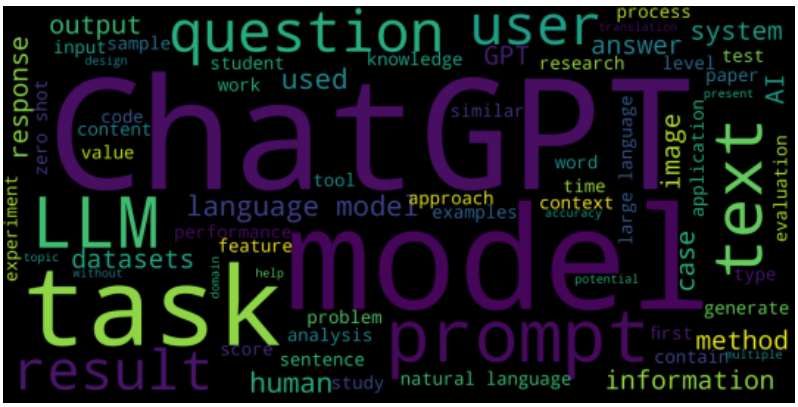


Figura 3. Nube de palabras sobre los 194 *papers* analizados por un estudio.
Fuente: https://arxiv.org/pdf/2304.01852.pdf.

Actualmente, los LLMs empiezan a explorar diferentes campos donde pueden ser aplicados según varios estudios. Algunos ejemplos son la educación, la traducción de textos, la generación de textos y, como se ha mencionado en todo momento durante este documento, en la generación de código.

En cuanto a la educación, puede tener una implicación indirecta sobre este estudio, ya que los LLMs tienen la capacidad de comprender un código dado o generado por ellos mismos y explicarlo con lenguaje natural. Es decir, aprovechando los conocimientos que tiene, es capaz de explicar el código que él mismo ha generado, en caso de que sea necesario y, además, advertir de los posibles problemas o dificultades que pueden surgir. Además, en caso de que se solicite validar el código generado con una colección de tests, dado que dispone de conocimientos para entender qué representa el código generado, también debe ser capaz de proporcionar tests que cubran todos los escenarios que pueden ocurrir en un código.

Por otro lado, la generación de textos también tiene implicaciones directas sobre la generación de código. Si bien es cierto que el conjunto de entrenamiento de la mayoría de los LLMs no es completamente código fuente, también es capaz de generar código, ya que, al fin y al cabo, el código no deja de ser texto con una sintaxis y lógica determinada. Adicionalmente, dentro del campo de la traducción de textos, se encuentra una funcionalidad que puede ser extremadamente útil dentro de la generación de código. Si bien esta peculiaridad está más enfocada en traducir textos escritos, como podría ser esta misma memoria, los LLMs también tienen la capacidad de poder traducir código de un lenguaje de programación a otro.

Dentro del ámbito de la generación de código, también ha habido grandes avances con resultados muy positivos. Concretamente, un estudio[6] ha podido constatar que usar tests para validar el código generado por un LLM, como podría ser GPT-3.5, mejora considerablemente los resultados. Además, si esta metodología se complementa con interacciones por parte del usuario que requiere el código (*active prompting*), la mejora es aún más notable. El estudio mencionado ofrece cuál es el grado de mejora del código generado en función de la cantidad de iteraciones en las que el usuario interviene con indicaciones para mejorarlo.

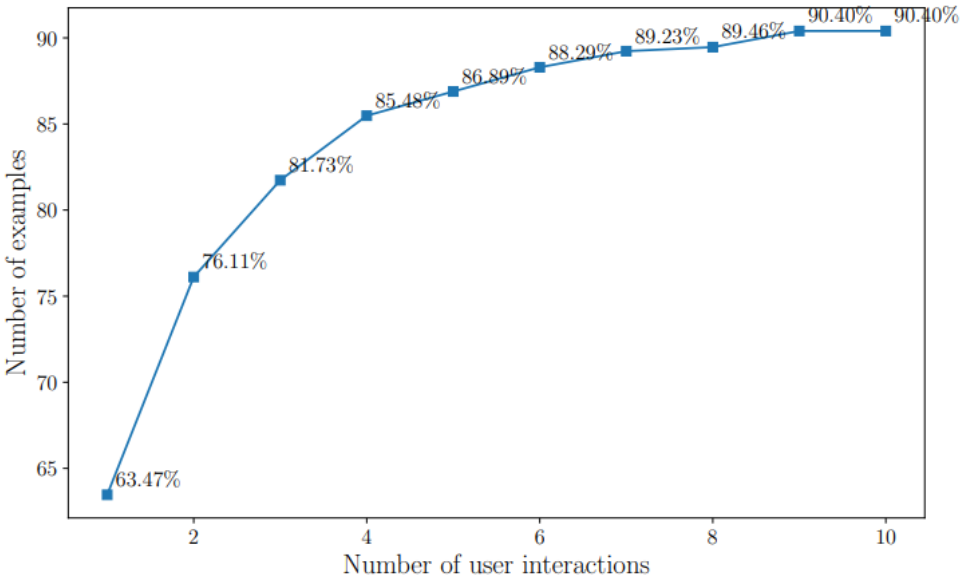


Figura 4. Porcentaje de códigos fuente que superan los tests en función del número de iteraciones. Fuente: https://arxiv.org/pdf/2208.05950.pdf.

Consecuentemente, el uso de tests para validar el código generado con indicaciones por parte de la persona que solicita el código ofrece buenos resultados. Sin embargo, cabe destacar que cada iteración implica hacer una llamada al LLM, por lo que, aunque los resultados son buenos a partir de un número de iteraciones, la eficiencia parece un aspecto que necesita mejora.

Otros estudios más recientes[7] también apuntan en la misma dirección. En el artículo citado, se utiliza la validación mediante tests creados por el propio LLM para validar el código fuente generado. Su estudio se centra en crear la mejor colección de tests posibles utilizando un nuevo modelo creado, *EvalPlus*, para poder verificar que el código generado es correcto. Es decir, no han enfocado tanto su estudio en intentar mejorar el código generado, sino en mejorar la calidad de los tests para validar con mayor certeza la validez del código. Por lo tanto, su investigación busca mejorar el determinismo que pueda tener la generación de código con un LLM. De hecho, uno de los problemas que aparece en la mayoría de los estudios, es que no hay una confianza ciega sobre si el código generado es correcto o no.

Continuando con el punto anterior, parece que aplicar TDD es una práctica bastante habitual. Ahora bien, los tests creados deben ser ejecutados por el mismo LLM. Esto puede llevar a que no simule correctamente el código de los tests y provoque falsos positivos, ya que el LLM no compila el código realmente. A partir de esta necesidad, nace la idea de poder dotar a un LLM con la capacidad de utilizar otras herramientas, como podría ser un compilador.

Una reciente publicación[5] sugiere usar un LLM como controlador de un modelo, concretamente ChatGPT, para poder resolver problemas enfocados a inteligencia artificial. Es decir, utilizan el mismo LLM para delegar en otros modelos o en él mismo otras tareas para poder resolver el problema inicial. Consecuentemente, la tendencia que se está buscando es dotar al LLM con la

capacidad de poder delegar en otros modelos para ofrecer una mejor respuesta.

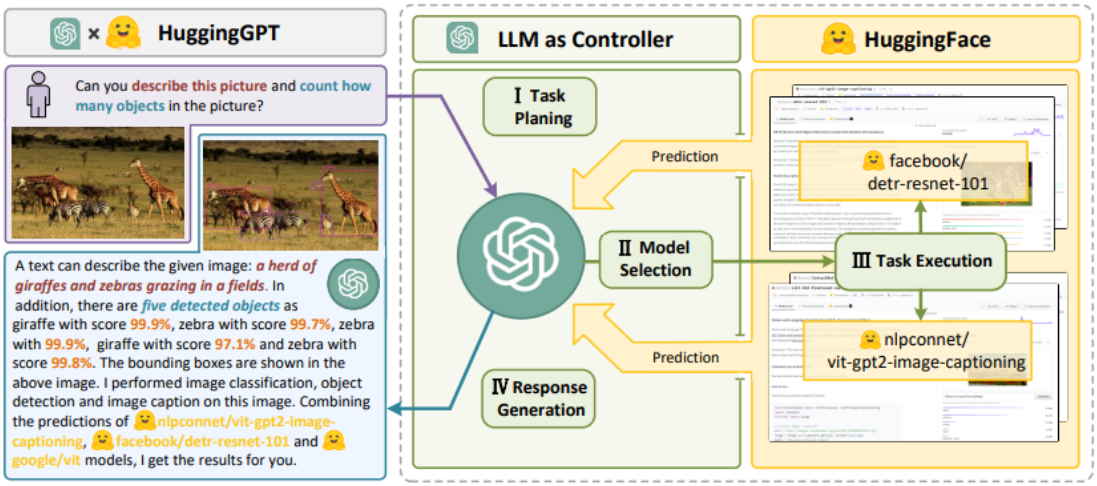


Figura 5. Esquema que representa el uso de un LLM como controlador de varios modelos en la tarea de delegar. Fuente: https://arxiv.org/pdf/2303.17580.pdf.

No obstante, el mismo *paper* deja claro que los resultados obtenidos hasta ahora no han sido los deseados, ya que existen algunas limitaciones que también se encontrarán en este estudio. Dependiendo del tipo de tarea que se le pide, la dependencia sobre la capacidad que tiene el LLM de descomponerla en otras tareas o en la habilidad de delegar, hace que los resultados no siempre sean los esperados. Por este motivo, esto provoca que el modelo se vuelva inestable. Además, dado que se tienen que hacer múltiples llamadas a varios modelos, se detectan problemas de eficiencia importantes. Finalmente, uno de los otros problemas que cita la investigación en este tipo de modelos es la limitación de *tokens* que existe.

Aunque en la publicación citada no habla sobre la aplicación de esta metodología en la generación de código, se puede extrapolar la idea e intentar aplicarla en este campo. Enlazando la propuesta anterior con la generación de código, si el LLM tiene la capacidad de usar un compilador para poder ejecutar el código con la colección de tests creados, se lograría un determinismo que antes no se tenía. Además, se podría determinar si el código generado es compilable o no.

En cualquier caso, como ya se ha dicho, este enfoque resulta inviable en el momento en el que se ha llevado a cabo este estudio debido a los problemas de eficiencia que pueden conllevar e incluso que la limitación de *tokens* sea un problema en la actualidad.

Finalmente, otro estudio que aporta un punto de vista diferente en la generación de código haciendo llamadas a un LLM presenta la herramienta GPTCOMCARE[9]. Concretamente, esta herramienta se encarga de desarrollar *n* veces un mismo código fuente con el objetivo de que la persona que ha solicitado el código pueda elegir el que tenga mejor calidad. Además, utiliza colores para poder ver a simple vista cuáles son las diferencias entre las diferentes versiones del código.

```
inputs ← GPT-n.responses;
n ← inputs.length;
for i ← 1, n do
    diffs ← [];
    for j ← 1, n do
        diff ← compare(inputs[i], inputs[j]);
        diffs.append(diff);
    end
    m ← diffs[0].length;
    for k ← 1, m do
        uniqueness ← 0;
        foreach diff ∈ diffs do
            if diff[k].startswith(+) then
                uniqueness += 1;
            end
        end
        if uniqueness > 0 then
            output += diff[k]
            (color: 127 + uniqueness * 32);
        else
            output += diff[k];
        end
    end
end
```

Figura 6. Algoritmo de la herramienta GPTCOMCARE que genera *n* versiones de un mismo código con el objetivo de poder compararlas. Fuente: https://arxiv.org/pdf/2301.12169.pdf.

En resumen, después de analizar múltiples estudios, los resultados obtenidos en la generación de código son buenos, pero no suficiente para poder delegar la generación del código a un LLM de manera no supervisada. Adicionalmente, cuando se delega al LLM la capacidad de poder gestionar otros modelos, mejora considerablemente los resultados. Ahora bien, dado que debe llevar a cabo más iteraciones, empeora la eficiencia y se llega a unos tiempos de espera considerablemente largos.

2.2. Herramientas actuales

Tal como se ha dicho en el punto anterior, ChatGPT es el modelo más popularizado actualmente. No obstante, existen otros LLMs que también son muy utilizados, incluso están presenten en entornos profesionales, como GitHub Copilot o IntelliCode Compose. En este apartado, por lo tanto, se contextualizan estas tres herramientas mencionadas haciendo énfasis en aquellas funcionalidades y características más relevantes para poder entender su relevancia dentro del mercado mientras se comparan entre ellas.

2.2.1. ChatGPT

Es la herramienta de la que se ha oído hablar más en los últimos meses. Es necesario especificar que ChatGPT no es un LLM per se, sino que es una plataforma presentada en forma de chatbot que utiliza un LLM para proporcionar respuestas. Sin embargo, se ha usado este término a lo largo de la memoria como si se tratara de un LLM por comodidad. Consecuentemente, cuando es tratado como un LLM, se hace referencia al LLM que utiliza y no a la herramienta conversacional.

Al inicio de este estudio, la única versión del LLM que estaba disponible era GPT-3.5. Actualmente, dispone de diferentes versiones, siendo la 3.5 gratuita y, en caso de tener una suscripción activada, se tiene la misma versión 3.5, pero con una velocidad de respuesta mucho más alta, y da acceso al LLM GPT-4. Incluso, si se ha logrado acceder a través de una lista de espera, se cuenta con una versión que permite buscar información por Internet, es decir, puede llegar a tener conocimiento en tiempo real y, además, también se ofrece una versión del mismo GPT-4, pero con *plugins*.[11]

Esta última funcionalidad es especialmente interesante si se relaciona con la generación de código. Como se mencionó anteriormente, los LLM pueden mejorar considerablemente la calidad del código generado si son capaces de acceder a otros modelos. Por lo tanto, actualmente se ofrece la posibilidad de crear un *plugin* el cual sea llamado justo después de crear el código y que este sea compilado por un compilador real y, además, que ejecute los tests que el mismo LLM ha creado.

Adicionalmente, también se cuenta con una API donde poder hacer llamadas desde modelos externos. Lamentablemente, solo se pueden hacer llamadas al modelo gpt-3.5-turbo y, en caso de haber accedido a través de la lista de espera, también se puede hacer uso del LLM gpt-4.

Recuperando el punto que trataba la limitación de *tokens* en el apartado anterior, hay que hacer una mención especial en esta herramienta. Actualmente, los modelos entrenados permiten recibir hasta 16.000 *tokens* aproximadamente[8]. Justo unos días antes de la entrega final de este documento se produjo un aumento, dotando a las nuevas versiones de cuatro veces más de longitud en el contexto. Es decir, el mensaje que se puede enviar puede ser cuatro veces más grande.

Asimismo, según la publicación referenciada, también se permite la opción de utilizar modelos externos. Por tanto, la tendencia que también sigue el mismo equipo de OpenAI es la que persiguen los *papers* que se han referenciado anteriormente: permitir que el LLM sea capaz de gestionar otros modelos según las necesidades que considere.

En cualquier caso, hay que tener en cuenta que ChatGPT es una herramienta que no ha sido diseñada exclusivamente para generar código (a diferencia de otras), sino que permite llevar a cabo muchas otras tareas. Por lo tanto, no utiliza un LLM que esté enfocado exclusivamente a la generación de código. Por un lado, esta característica puede parecer un problema, ya que puede dar a entender que no tiene un nivel de experticia adecuado para generar código correctamente. Sin embargo, como se demuestra en todos los estudios que se han comentado anteriormente, este razonamiento no es correcto. Todo depende del *prompt* que recibe, es decir, del contexto que se le da.

2.2.2. GitHub Copilot

Esta herramienta está pensada para generar código fuente, aunque su conjunto de entrenamiento no es solo código. El LLM que utiliza se llama Codex y pertenece a OpenAI y GitHub. Concretamente, es un LLM que parte del modelo entrenado gpt-3[10]. Hoy en día, solo está disponible con la versión de pago que ofrece GitHub o de forma gratuita en la versión para estudiantes y profesores[12].

La misma documentación de esta herramienta ya proporciona instrucciones sobre cómo se puede integrar dentro de un IDE. Además, permite la generación de código a partir de *prompts*, al igual que ocurre con ChatGPT, pero, además, también permite autocompletar código automáticamente en función del inicio del código que se está desarrollando. Esta funcionalidad no solo está entrenada con un conjunto de entrenamiento externo al usuario final, sino que también se entrena a partir del código que se desarrolla. Es decir, tiene la capacidad de tener memoria en función de lo que se ha escrito previamente e intentar predecir si quiere volver a poner lo mismo o algo similar.

Actualmente, la limitación de *tokens* que tiene este LLM es la mitad de lo que permite ChatGPT con la última actualización. Además, esta herramienta no permite probar el código automáticamente como si se puede hacer con ChatGPT. Es decir, el LLM devuelve el código generado en función de un *prompt*, pero, en caso de querer comprobar si el código es correcto, se debe volver a crear un nuevo *prompt* (o utilizar el anterior) y especificarle que desarrolle tests para validar el comportamiento del código generado. Consecuentemente, la validación del código debería hacerse en varios pasos.

Finalmente, OpenAI desactivó el acceso a la API del LLM que utiliza esta herramienta, Codex. Por lo tanto, no se pueden hacer llamadas automáticamente a la API de este LLM[13].

2.2.3. IntelliCode Compose

Este LLM creado por Microsoft tiene ligeras diferencias respecto a IntelliSense. Este último ofrece sugerencias de autocompletado de código. En cambio, IntelliCode va más allá y hace que las sugerencias apliquen técnicas de inteligencia artificial en función de cómo se ha comenzado la línea de código. De esta manera, se dota a la herramienta con un mejor porcentaje de predicción y mejora la eficiencia de la persona que programa.

Sin embargo, es importante destacar que esta función no está disponible para todos los lenguajes de programación. Concretamente, esta herramienta solo está disponible para C#, XAML, C++, JavaScript, TypeScript y Visual Basic.[14]

Como se puede observar, esta herramienta no ofrece la posibilidad de generar código a partir de lenguaje natural. Por lo tanto, es

considerablemente diferente a las otras dos mencionadas anteriormente. No obstante, considerando su funcionamiento interno y el alto uso de esta, es relevante incluirla en este capítulo.

Para terminar, es importante mencionar que esta herramienta no es accesible mediante alguna API y solo está disponible de forma gratuita a través de los IDE oficiales de Microsoft como Visual Studio y Visual Studio Code.

2.3. Metodologías de estudios actuales

Tal como se ha expuesto en el primer capítulo de esta memoria, este estudio busca evaluar la generación de código aplicando diversas metodologías. Consecuentemente, es necesario contextualizar el estado actual de estas metodologías dentro de la generación de código, ya que, como se verá a continuación, no solo se utilizan para generar código.

2.3.1. Test Driven Development

La metodología TDD tiene un largo recorrido dentro del desarrollo de software. Concretamente, ha sido explorada desde hace casi 50 años. En ella, la colección de tests es la que determina la validez del código desarrollado. Es decir, se dispone de un conjunto de tests que cubran todos los requisitos de un sistema o aplicación y es el código por desarrollar el que se debe construir con el objetivo de superar estos tests.

Esta práctica también se ha visto presente en varios estudios enfocados en la generación de código utilizando LLMs, ya que precisamente permite determinar si el software generado se ajusta a las necesidades definidas en la colección de tests. En consecuencia, puede convertirse en un elemento que ofrezca el determinismo que se exige a cualquier modelo para brindar con confianza. Ahora bien, si la colección de tests también es generada usando un LLM, el problema no queda resuelto, sino que se traslada a otra capa del modelo.

Con el objetivo de mitigar este problema, hace unos meses se publicó un estudio donde presentaban la herramienta LIBRO[15]. Este modelo utiliza los LLMs para la generación de tests a partir de una entrada que proporciona un usuario. Así, cuando se detecta un *bug*, este es reportado e introducido dentro de un *prompt* junto con un archivo de tests como ejemplo. Una vez realizada la llamada al LLM con toda esta entrada, este genera uno o varios ficheros con tests que deben cubrir todos los escenarios dados a raíz del *bug* reportado, con el objetivo de después poder desarrollar el código para superar la colección de tests proporcionada por el LLM y, finalmente, considerar corregido el error.

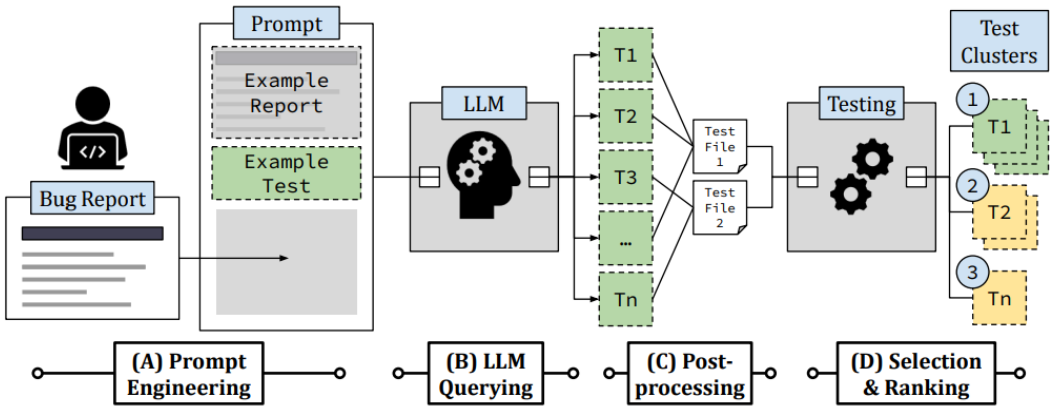


Figura 7. Representación conceptual del modelo LIBRO.
Fuente: https://abhikrc.com/pdf/ICSE23.pdf.

Con este modelo, se ha demostrado que se mitiga la problemática anterior considerablemente, ya que permite generar múltiples archivos con tests en función de los ficheros de ejemplo con el objetivo de que el programador solucione los errores en el código fuente del programa para superar las pruebas. Sin embargo, el estudio llevado a cabo advierte sobre ciertos comportamientos que ocurren. Uno de estos se refiere al número de tests que debe generar el LLM. Concluyen que hay algunos errores que, aunque se aumente considerablemente el número de tests a generar, no necesariamente quedarán cubiertos todos los escenarios. Por lo tanto, **un gran número de tests generados no implica necesariamente que todos los requisitos están cubiertos**.

Haciendo énfasis en el mismo apartado, uno de los estudios ya citados anteriormente[7], también utilizaba esta metodología para determinar si el código generado con el LLM es validado, o no, por los tests. Concretamente, primero generan una colección de tests de calidad para luego ejecutarla con el código generado. Es decir, llevan a cabo llamadas sobre el mismo LLM para generar los tests con los datos necesarios para cubrir el máximo de escenarios posibles. Adicionalmente, también generan una colección más pequeña que garantice el funcionamiento mínimo del código con el objetivo de mejorar la eficiencia del modelo. En cualquier caso, este modelo no genera código por él mismo, sino que se encarga de proporcionar una colección de tests en función del código que ya se haya generado anteriormente.

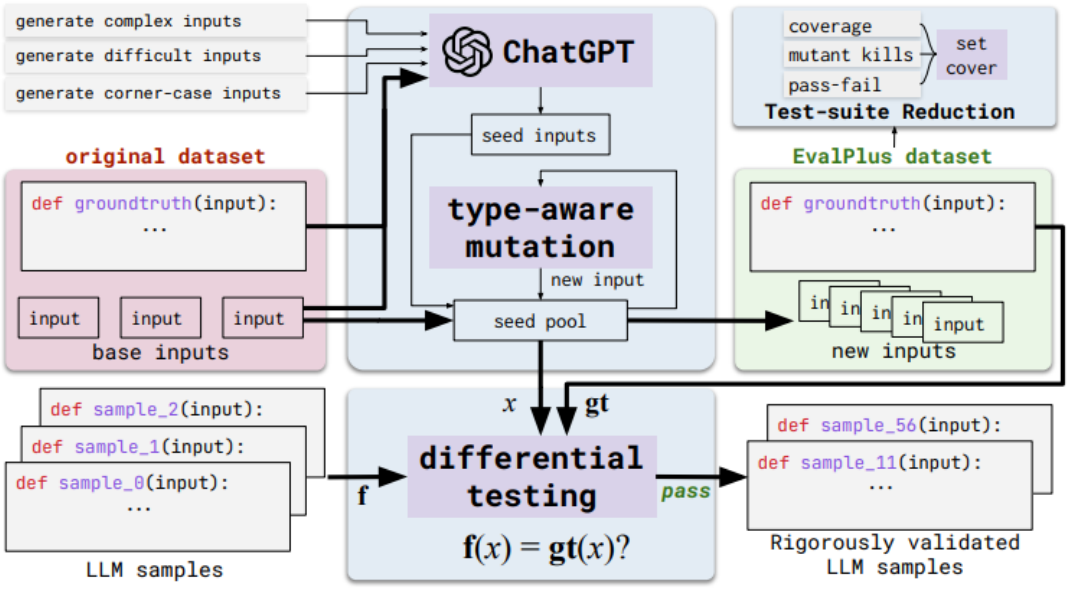


Figura 8. Representación conceptual del modelo *EvalPlus* que genera una colección de tests de calidad para validar el código generado por un LLM.
Fuente: https://arxiv.org/abs/2305.01210.

Consecuentemente, se puede afirmar que es una metodología que se está usando actualmente como validador en los generadores de código y es un camino que debe de continuar teniendo recorrido dentro de la generación de código con LLMs.

2.3.2. Chain-of-thought

Una de las otras metodologías que da muy buenos resultados con los LLMs es el *chain-of-thought*. Esta tiene como objetivo motivar el pensamiento crítico del LLM a partir de una cadena de pensamiento. Es decir, se busca descomponer un problema u objetivo complejo en varios pasos intermedios. De esta manera, el modelo escribe todos estos pasos y permite generar una respuesta más precisa y correcta. Teniendo en cuenta que un LLM es un grafo con varias capas que determina la respuesta que da en función de la entrada o lo que ha escrito, si se le obliga a escribir los pasos previos a esta respuesta final, el modelo es capaz de determinar con mucha más precisión cuál será la siguiente palabra que debe utilizar.

Esta metodología ya se ha encontrado en varios de los estudios que se han citado. Por ejemplo, en el estudio donde se presentaba el modelo LIBRO[15], se proporcionaba un archivo de tests como ejemplo. Si bien es cierto que en este caso el LLM no escribe pasos intermedios para llegar a la respuesta, sí tiene que leer la entrada y analizar qué tipo de respuesta se espera. Consecuentemente, se está induciendo un pensamiento crítico enfocado de una manera concreta para llegar a la respuesta deseada con mucha más precisión.

Adicionalmente, también se encuentran estudios recientes que analizan la considerable mejora de esta metodología aplicada a los LLMs. Por ejemplo, en un artículo publicado hace unos meses, evaluaban la mejora de las

respuestas dadas por un LLM induciendo la cadena de pensamiento. Además, esta metodología ofrece resultados considerablemente buenos en campos como la aritmética o el sentido común. Los LLMs tienen una gran capacidad de generación de texto, pero no aplican un pensamiento lógico. Por tanto, si se descomponen los problemas, la capacidad de aplicar un pensamiento lógico mejora. El estudio se ha llevado a cabo con varios LLMs y en todos mejoran mucho los resultados.

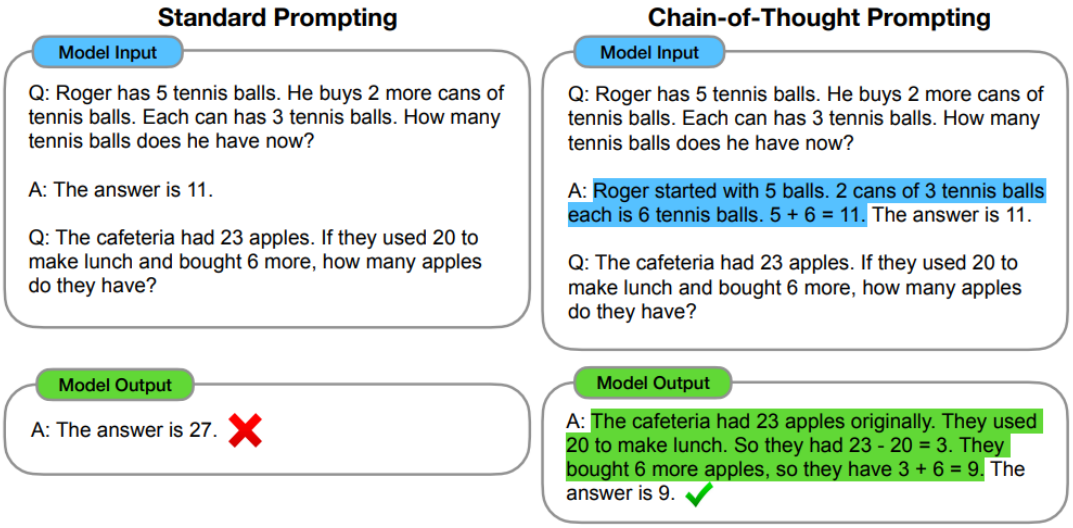


Figura 9. Comparación de las respuestas entre la aplicación y la no aplicación de la cadena de pensamiento. Fuente: https://arxiv.org/pdf/2201.11903.pdf.

En consecuencia, teniendo en cuenta que la generación de código contiene un alto grado de pensamiento lógico, **la aplicación de esta metodología parece un elemento clave para obtener mejores resultados** según los estudios mencionados.

2.3.3. *Active prompting*

Para terminar con las metodologías relevantes en la generación de código, es obligatorio tratar el *active prompting*. Este permite interactuar con el LLM y darle indicaciones sobre los cambios que debe hacer sobre una primera respuesta ya generada. Es decir, esta metodología se aplica una vez que ya se ha obtenido una primera respuesta, se evalúa y, en caso de ser necesario, se interactúa con el LLM.

Además, esta metodología permite mitigar los posibles errores que se hayan generado de manera manual. Tal como se está presentando en este capítulo, el código generado no siempre es correcto, de manera que con la intervención humana se pueden corregir los posibles errores o sesgos, es decir, mitigar los problemas mencionados anteriormente. En cualquier caso, esta metodología debe acompañar a las otras mencionadas para poder sacar todo su potencial. En otras palabras, aplicar *active prompting* sin una validación del código generado con tests, puede mejorar el resultado, pero es con la combinación de ambas cuando se llegan a mejores resultados.

Adicionalmente, si esta metodología también se aplica con la cadena de pensamiento mencionada anteriormente, los resultados también mejoran considerablemente. Un estudio reciente confirma esta afirmación[16]. En él, concluyen que, si los *prompts* dados por el usuario contienen preguntas útiles y se dan ejemplos efectivos en lugar de hacer preguntas arbitrarias o ejemplos poco relevantes, los resultados obtenidos serán mucho más precisos.

En consecuencia, la aplicación de esta metodología conjuntamente con las otras ya mencionadas enriquece el modelo produciendo respuestas mucho más precisas y, además, dota a un modelo con la capacidad de poder interactuar con él.

2.4. IDE con generadores de código

Actualmente, existen diversos IDE que ya incorporan generadores de código que aplican inteligencia artificial. De hecho, en este capítulo ya se han introducido algunos de ellos. Concretamente, se ha hablado del caso de IntelliCode Compose, donde se utiliza un LLM para predecir cuál será la siguiente palabra dentro del código de una manera mucho más precisa que con IntelliSense.

Ahora bien, las herramientas exclusivamente de Microsoft no son el único ejemplo, ya que existen otros LLMs como Codex que también se pueden utilizar. Tal como se ha dicho anteriormente, GitHub Copilot se puede añadir dentro de Visual Studio Code mediante una extensión. Y, de hecho, esta no es la única extensión que existe porque, dado que varios LLMs permiten hacer llamadas a la API, cualquier usuario puede crear su propia extensión y publicarla. Es decir, cualquier persona puede desarrollar una extensión para Visual Studio Code y hacer llamadas a la API del LLM que quieran usar.

Asimismo, también existen otros generadores de código incorporados dentro de los mismos IDE. Por ejemplo, JetBrains dispone de un generador de código en sus IDEs. A pesar de esto, este no hace llamadas a ningún LLM para generarlo, sino que se basa en el nombre de la clase y de los atributos (en caso de ser un lenguaje orientado a objetos) para generar constructores y los *getters* y *setters* necesarios. Por lo tanto, aunque hay IDEs que afirman tener generadores de código, no todos utilizan un LLM para hacerlo.

La previsión es que en el futuro esta área aumente considerablemente con el gran auge que están teniendo los LLMs. Sea con los LLMs que existen actualmente o con otros nuevos que puedan surgir, la gran cantidad de *papers* y los buenos resultados que se empiezan a obtener, hacen pensar que así será. En cualquier caso, en todos los generadores de código que hay en la actualidad se requiere la supervisión humana, es decir, no hay ningún generador de código en estos momentos que garantice que el código generado es correcto. Por lo tanto, estas herramientas deben entenderse como una ayuda al programador y una manera de agilizar el proceso de desarrollo de software. En ningún caso los estudios afirman que, hoy en día, se pueda sustituir la figura del desarrollador de código en entornos profesionales.

3. Diseño del prototipo

En este capítulo se tratará todo lo que hace referencia al diseño del prototipo sin entrar en detalles son el desarrollo de este.

3.1. Plataforma de ejecución

En una primera instancia, el prototipo se ha desarrollado pensando que la plataforma sobre la cual se ejecutaría sería Windows 10. Es decir, solo se realizarían llamadas a la API del LLM localmente. Ahora bien, teniendo en cuenta que la intención del trabajo es también poder ofrecer una extensión dentro de Visual Studio Code, se debe permitir la ejecución remotamente desde cualquier servidor en la nube. Esto implica que se pueden realizar consultas al prototipo alojado en un servidor y que este devuelva la respuesta que ha proporcionado el LLM, debidamente procesada.

Así pues, el prototipo debe ser multiplataforma y se debe poder ejecutar desde Windows para poder llevar a cabo todas las pruebas necesarias y, además, debe poder ejecutarse en un servidor de una distribución de Linux. En este caso, se ha considerado que sea una distribución de Linux porque son los sistemas operativos con más disponibilidad dentro de las principales empresas de alojamiento. Además, la compatibilidad de las aplicaciones entre las diferentes versiones de Linux es muy alta. Por lo tanto, se disminuye el riesgo de que no pueda ejecutarse adecuadamente.

En conclusión, con el escenario planteado, se cubren todas las necesidades que tiene el proyecto para poder llevarse a cabo satisfactoriamente.

3.2. Justificación del lenguaje escogido

Se ha decidido que el lenguaje con el cual se desarrollará todo el prototipo sea Python. Los motivos por los cuales Python es el lenguaje correcto para llevar a cabo este proyecto son varios:

- Bibliotecas disponibles: Python dispone de una gran cantidad de bibliotecas públicas que permiten desarrollar ciertas funcionalidades en un tiempo mucho menor al que se tardaría haciéndolo íntegramente. Adicionalmente, muchas de estas bibliotecas están enfocadas precisamente en el uso de inteligencia artificial y, más concretamente, en el uso de LLMs. De hecho, existe una biblioteca basada en Python llamada *LangChain*[17] que permite hacer llamadas a una API de un LLM y gestionar las respuestas de una manera muy sencilla. Además, también permite crear cadenas con el mismo LLM, dando la posibilidad de mantener una conversación con memoria. Más adelante se hablará en detalle de esta y otras bibliotecas que se han utilizado para desarrollar el prototipo.

- Alta compatibilidad: es justo uno de los requisitos según el apartado anterior, ya que se tiene que poder ejecutar localmente (en un entorno con Windows 10) y en un servidor en la nube. De hecho, Python es un lenguaje de programación que permite instalarse de manera simple a todos los entornos y plataformas de ejecución que se han mencionado anteriormente. Consecuentemente, también resulta un lenguaje adecuado en este sentido.
- Facilidad a la hora de trabajar con datos: esta peculiaridad está relacionada con la primera, ya que Python ofrece un amplio abanico de bibliotecas que permiten trabajar con diferentes tipos de datos. Aunque aún no se ha especificado, el prototipo necesitará gestionar las respuestas recibidas en formato XML. Esta característica del prototipo se detallará en el siguiente capítulo, pero es necesario mencionarla para poder entender la necesidad de trabajar fácilmente con diferentes tipos de datos.
- Comunidad grande y activa: Python es uno de los lenguajes de programación más utilizados del mundo, si no el que más[18]. Gracias a su simplicidad, legibilidad, escalabilidad y compatibilidad, se convierte en una de las opciones preferidas de muchos desarrolladores. Consecuentemente, la comunidad vinculada a este lenguaje es una de las más grandes, lo que repercute positivamente sobre cualquier proyecto desarrollado en Python, ya que, en caso de que surjan problemas durante el desarrollo, es muy probable que se pueda encontrar la solución a este rápidamente.

Por lo tanto, teniendo en cuenta todas las ventajas que ofrece desarrollar el prototipo en Python según el contexto requerido por el proyecto, la elección queda totalmente justificada.

3.3. LLMs utilizados por el prototipo

Tal como se ha introducido anteriormente, el estudio solo se llevará a cabo utilizando un solo LLM, concretamente, gpt-3.5-turbo de OpenAI. Sin embargo, el prototipo deberá diseñarse manteniendo una escalabilidad en este aspecto. Es decir, se deberán minimizar los cambios a realizar entre un LLM y otro.

Haciendo énfasis sobre la decisión de usar un solo LLM, esta se ha tenido que adoptar por varios motivos.

El primero tiene que ver con los plazos de este estudio. Teniendo en cuenta todas las metodologías que se quieren evaluar, resultaría excesivo llevar a cabo el estudio con múltiples LLMs considerando la planificación del proyecto.

En segundo lugar, reafirmando el primer motivo, durante la planificación del trabajo solo había disponible un LLM en la API de OpenAI. Justo unas semanas antes se deshabilitaron las llamadas a la API de Codex y para poder acceder a la API de GPT4 se tiene que hacer mediante una lista de espera, lo que hace que la idea inicial de hacerlo con múltiples LLMs se vuelva imposible.

En resumen, el estudio solo hará llamadas a la API de un solo LLM aplicando múltiples metodologías tal como se ha explicado en el primer capítulo.

3.4. Diseño conceptual del prototipo

Con el objetivo de tener una planificación que garantice una correcta escalabilidad del prototipo, se ha llevado a cabo un diseño conceptual de este. Concretamente, se ha diseñado el prototipo en UML (*Unified Modeling Language*), donde quedan representadas todas las metodologías que se quieren aplicar. Además, también se gestiona de manera independiente la lectura de la documentación en formato *javadoc*, ya que se considera que es una funcionalidad que solo debe tener el prototipo con el cual se ejecutan las pruebas.

Así, pues, la siguiente figura corresponde al diagrama UML que representa el prototipo en un ámbito de clases.

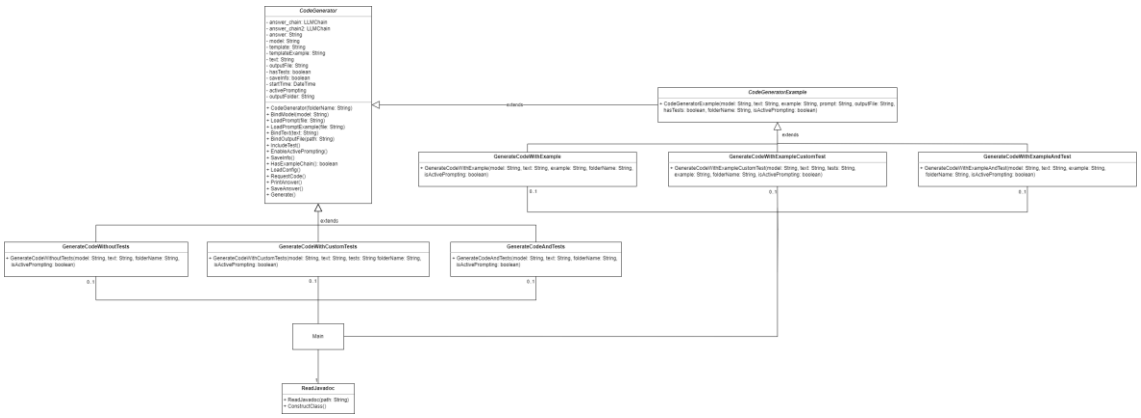


Figura 10. Diagrama UML que representa el diseño del prototipo. Fuente: original.

Con el objetivo de proporcionar una mejor visualización de este diagrama, se ha adjuntado un enlace a un sitio web en el Anexo V, donde se puede consultar en formato de imagen.

Como se puede observar en el modelo propuesto, hay una clase *CodeGenerator* que es la encargada de gestionar todo el proceso de hacer la llamada al LLM y mostrar y almacenar los resultados.

Por un lado, a partir de esta clase padre, se han creado tres clases hijas instanciables que permiten generar el código con las tres metodologías propuestas anteriormente: sin tests, con tests dados y con la creación de tests por el mismo LLM.

Por otro lado, se ha creado un nuevo nivel de herencia que permite generalizar el comportamiento de la generación de código utilizando una cadena de *LangChain* donde se da un ejemplo de lo que debe crear antes de solicitar el código. Por lo tanto, nuevamente aparecen tres clases instanciables que permiten hacer las llamadas con diferentes metodologías aportando un ejemplo para cada una de ellas.

En cuanto a la metodología de *active prompting*, se ha diseñado a partir de uno de los parámetros del constructor de cada una de las clases instanciables.

Finalmente, es necesaria una clase que permita leer el contenido de los archivos *javadoc* para poder enviar esta información al LLM.

3.5. Ejecución y funcionamiento

Considerando que el prototipo se crea para poder llevar a cabo un análisis sobre las diferentes metodologías para generar código y no es un producto final que deba entrar en producción, la ejecución de este debe ser lo más simple posible. Consecuentemente, se ha decidido que el prototipo se ejecute mediante la línea de comandos.

Además, dado que se deben poder ejecutar diversas metodologías, el prototipo debe ofrecer esta posibilidad de manera directa. Para hacerlo, se utilizarán varios argumentos que permitan ejecutar el prototipo de una forma u otra.

La sintaxis que se utilizará para ejecutar el prototipo debe ser la siguiente:

py __main__.py 1|*2*|*3*|*example1*|*example2*|*example3*|*all*|*times*[*active*]

Donde:

- *1*: genera el código sin validarlo con tests.
- *2*: genera el código y lo valida con la colección de tests dada.
- *3*: genera el código y lo valida con los tests creados por él mismo.
- *example1*: igual que la opción "*1*", pero dando un ejemplo previo.
- *example2*: igual que la opción "*2*", pero dando un ejemplo previo.
- *example3*: igual que la opción "*3*", pero dando un ejemplo previo.
- *all*: ejecuta la opción "*1*", "*2*" y "*3*" en cadena.
- *times*: ejecuta la opción "*all*" *n* veces. Por defecto, se ha fijado a 10 veces.

Por lo tanto, un ejemplo de ejecución en la línea de comandos debería tener un formato como el siguiente:

py __main__.py 3 active

Teniendo en cuenta en todo momento que se está ejecutando el comando desde el directorio donde reside el prototipo y que el archivo *__main__.py* contiene el punto de entrada al prototipo. En este caso, se estaría ejecutando el prototipo con la generación de tests por el mismo LLM y con la posibilidad de aplicar *active prompting* y volver a enviar un nuevo *prompt* a la API.

En resumen, toda la ejecución del prototipo se debe poder llevar a cabo mediante la línea de comandos. Ahora bien, en caso de que haya alguna metodología que no ofrezca los resultados esperados, no será necesario preparar el prototipo con la ejecución de esta metodología por la línea de comandos.

4. Desarrollo y ejecución del prototipo

Una vez se ha finalizado el diseño conceptual del prototipo y se han definido todas las necesidades de este, se ha detallado cómo ha sido el desarrollo del prototipo.

4.1. Preparación del entorno de desarrollo

Antes de empezar con el desarrollo, se ha tenido que preparar el entorno de trabajo con diversas herramientas. Adicionalmente, para poder desarrollar el prototipo, se debe disponer de un IDE. En este caso, se ha seleccionado Visual Studio Code por su comodidad, ya que ofrece una terminal integrada y permite instalar fácilmente diversos *plugins*.

4.1.1. Instalación de los lenguajes necesarios

Como se ha mencionado anteriormente, el lenguaje de programación utilizado es Python. Por lo tanto, dado que no es necesario el uso de ningún otro lenguaje para llevar a cabo el desarrollo, se ha instalado una de las últimas versiones estables de este lenguaje, la 3.10.4.

4.1.2. Configuración con la API

Con el objetivo de poder hacer llamadas a la API de OpenAI, es necesario configurar una clave privada que identifica al usuario que está realizando las llamadas a esta. Para hacerlo, se ha tenido que crear una clave privada dentro de la plataforma que ofrece OpenAI.

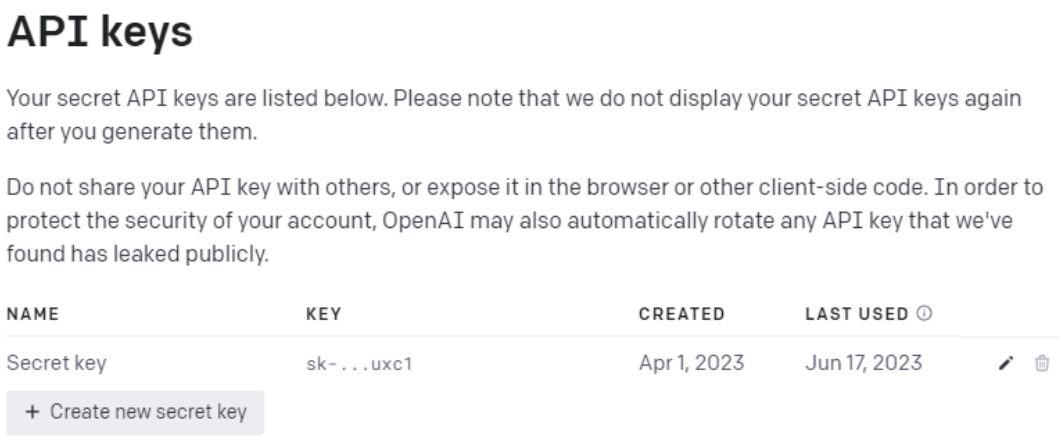


Figura 11. Captura de pantalla de la plataforma de OpenAI donde se ha creado la clave privada. Fuente: original.

Una vez la clave ha sido creada, se ha almacenado en un archivo llamado *.env* donde hay variables de entorno utilizadas en el prototipo. Este se puede consultar en el mismo repositorio que se ha adjuntado en el Anexo IV.

4.1.3. Biblioteca utilizada para generar y validar código

Desde el inicio del trabajo, se ha planificado en todo momento que se trabajaría con la biblioteca *LangChain*, ya que ofrece mecanismos para llevar a cabo llamadas a las APIs de varios LLMs de una manera muy fácil y sencilla. Por lo tanto, ha sido necesario instalar la biblioteca e importarla dentro del proyecto.

Adicionalmente, también se ha trabajado con otra biblioteca llamada *GuardRails*. No obstante, finalmente ha tenido que ser descartada porque provocaba tiempos de espera demasiado largos en algunas situaciones. Esto es debido a que la biblioteca mencionada verifica que la respuesta que se recibe cumpla con un conjunto de normas definidas. Por consiguiente, si no las cumple, vuelve a solicitar al LLM el mismo *prompt*, entrando en un bucle del cual puede llegar a ser difícil salir. Así, se ha optado por no utilizarla, ya que la eficiencia temporal del prototipo se volvía considerablemente inviable.

4.2. Repositorio del código fuente

Con el objetivo de facilitar el código fuente desarrollado y de poder mantener un versionado del mismo, se ha decidido crear un repositorio público para poder compartirlo de manera rápida, sencilla y profesional. En apartados anteriores ya se ha compartido el enlace de este repositorio. Concretamente, se puede acceder a él a través del Anexo IV.

4.3. Tratamiento de la respuesta del LLM

Como ya se introdujo en el capítulo anterior, se ha solicitado dentro del *prompt* que el LLM devuelva las respuestas en lenguaje XML. De esta manera, se consigue generalizar la respuesta que da para poder tratarla siempre de la misma forma.

Concretamente, se espera recibir dos etiquetas: *code* y *tests*. La primera debe contener el código fuente generado por el LLM, mientras que la segunda debe contener el resultado de todos los tests generados o enviados, si es que tiene alguno. Adicionalmente, cada una de los tests debe venir dentro de una etiqueta *test* con diferentes valores como atributos: el identificador del test, su resultado, el caso que evalúa, el valor esperado, el valor recibido y, finalmente, el motivo por el cual ha fallado, si ese fuera el caso.

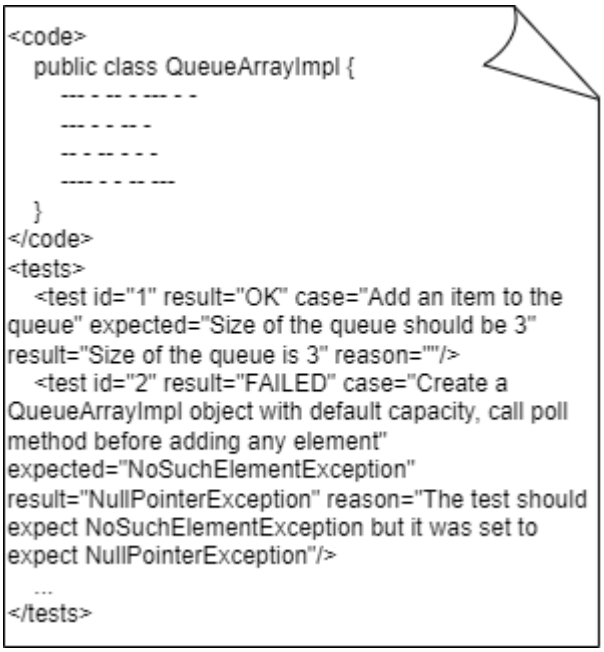


Figura 12. Ejemplo de una respuesta correcta con tests por parte del LLM. Fuente: original.

A pesar de haberlo especificado claramente en el *prompt*, inicialmente se detectaron situaciones donde el código generado no contenía las etiquetas necesarias para poder gestionar la respuesta recibida. Como ya se mencionó en la sección anterior, en un primer momento se intentó dar solución a este problema con la biblioteca *GuardRails*, pero finalmente se descartó por motivos de eficiencia.

Consecuentemente, como se detallará en las siguientes secciones, hay algunos casos donde la respuesta dada por el LLM no mantiene el diseño solicitado en el *prompt*.

Finalmente, cabe destacar que la respuesta, siempre que esta retornara código, siempre ha sido sintácticamente correcta, es decir, **no se ha observado ningún error sintáctico en el código generado**. Ahora bien, sí se han detectado errores que provocarían que el código no compilase en algunas situaciones. Por ejemplo, si no se indica específicamente, el LLM no incluye en algunas ocasiones la inclusión de las bibliotecas o clases necesarias. En cualquier caso, este hecho ocurre debido a la falta de contexto que tiene sobre el conjunto del software a generar.

4.4. Diseño de los *prompts*

Como ya se ha ido introduciendo a lo largo del trabajo, la generación de código se lleva a cabo enviando un *prompt* que contiene lo que se quiere generar utilizando lenguaje natural. Es decir, es necesario crear un archivo de texto que contenga las indicaciones para que el LLM sea capaz de entender lo que se está solicitando (el *prompt*).

Esta ha sido una fase en la que el método de prueba y error ha sido la metodología empleada para llegar al resultado final. En un principio, el *prompt*

escrito solicitaba al LLM que generase código como si fuese un texto natural. No obstante, después de diversas pruebas y analizar otros *papers*, se llegó a la conclusión de que la mejor manera para que un LLM entienda con precisión lo que se está solicitando no es utilizar este formato, sino que la mejor manera de hacerlo es mediante órdenes claras y enumeradas.

Para ello, se deben definir unos requisitos que debe cumplir la salida que genere el LLM. De esta forma, se le está indicando que se asegure de que la respuesta cumpla todos los puntos indicados.

Aunque de esta manera el resultado mejoraba considerablemente, en algunas ocasiones se han detectado reglas que no se cumplían. Por ejemplo, una de las reglas es que no debe generar comentarios dentro del código generado. Y aún así, había casos en los que añadía comentarios sobre la cabecera de los métodos. Este tipo de comportamientos no tienen explicación por ahora, ya que incluso los creadores del LLM utilizado tampoco entienden cómo llegan a ciertas conclusiones.

Adicionalmente, otro aspecto totalmente imprescindible que debe tener el *prompt* es una definición clara y concisa del rol que debe adoptar en todo momento. Concretamente, como la intención es generar código y poder validarlo, se le debe decir que debe actuar como un generador y validador de código. Es decir, solo puede proporcionar código y los resultados de los tests, si es que los tiene. Esta parte es especialmente importante en un LLM como GPT, ya que este tiende fácilmente a dar explicaciones sobre cuál es el procedimiento que sigue o cuál es la justificación de lo que ha hecho.

Relacionado con el aspecto anterior, si se limita el tipo de respuesta que puede dar, se puede llegar a conseguir que las respuestas estén generalizadas, es decir, que siempre sigan un mismo formato. Además, para poder garantizar que se pueda tratar la respuesta recibida, dentro del mismo *prompt* se ha especificado en qué formato debe generar el texto. En este caso, se le ha pedido que genere el código dentro de una etiqueta XML llamada *code* y que enumere todos los tests generados dentro de otra etiqueta llamada *tests*. Como resultado, la gestión de la respuesta es más sencilla porque siempre mantiene el mismo formato.

A continuación, se debe mencionar otro hecho que se ha detectado durante el diseño de los *prompts*. Cuando estos son enviados, el LLM genera una respuesta. En principio, si el *prompt* es el mismo y el atributo *temperature* está asignado a 0, el mínimo posible, debería devolver siempre la misma respuesta. Sorprendentemente, este no ha sido el caso, haciendo que el código generado (y los tests) raramente sean igual entre las distintas llamadas al LLM. El escenario mencionado tendría sentido si se hubiese asignado un valor superior a 0 en la *temperature*, pero había aleatoriedad en las respuestas incluso con este tipo de configuración. Por lo tanto, **aunque se asigne el valor mínimo al atributo *temperature*, siempre existe un cierto componente de aleatoriedad en la generación de la respuesta**.

En diversos estudios ya citados en este documento[7][15], se han llevado a cabo evaluaciones con diferentes valores para este parámetro, pero en este estudio se ha querido poner a prueba la generación de código con el mínimo de aleatoriedad posible porque se entiende que el componente de aleatoriedad en las respuestas puede ser un problema y provocar escenarios no deseados con más facilidad.

En resumen, la mejor estructura que debe tener un *prompt* según las múltiples pruebas que se han llevado a cabo, teniendo en cuenta que es muy importante el qué y el cómo se coloca, es la siguiente:

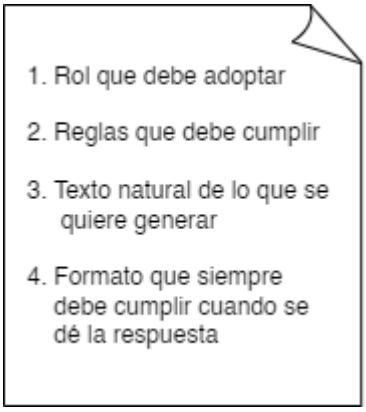


Figura 13. Esquema del contenido y el orden del *prompt* generado durante el estudio. Fuente: original.

4.5. Generación de código a partir de lenguaje natural

Esta metodología corresponde a la primera de las metodologías descritas en la introducción. Concretamente, a partir del nombre de la clase y de sus clases heredadas, la cabecera de los métodos que se quieren generar y el comentario asociado a cada uno de estos métodos, se solicita al LLM que genere todo el cuerpo de los constructores y de los métodos y declare los atributos necesarios para poder satisfacer los requisitos solicitados en los comentarios.

Durante la generación de código utilizando esta metodología, se ha detectado que es cuando el resultado que devuelve el LLM no coincide en más ocasiones con el formato solicitado. Es decir, cuando no se crean tests que validen el resultado, se provocan situaciones en las que, por ejemplo, los métodos no tienen cuerpo.

Ahora bien, estas situaciones se han logrado mitigar modificando los *prompts*, sin embargo, cuando el contexto dado es considerablemente largo, la probabilidad de que esto suceda aumenta.

```java

public class LinkedList<E> extends Object implements List<E> {

  protected int count; // number of elements in the list
  protected LinkedList.LinkedNode<E> head; // first node in the list

  // Empty list.
  public LinkedList() {
    count = 0;
    head = null;
  }

  // Delete received position.
  public E delete(Position<E> node) {
    // code here
  }

  // Delete the first position in the list.
  public E deleteFirst() {
    // code here
  }

  // Delete the next position.
  public E deleteNext(Position<E> node) {
    // code here
  }

  // Add an item after the received position.
  public Position<E> insertAfter(Position<E> node, E elem) {
    // code here
  }
```

Figura 14. Captura de pantalla de una parte de la clase generada con esta metodología donde los métodos no tienen cuerpo. Fuente: original.

Adicionalmente, como se puede observar, ha añadido un comentario dentro de cada uno de los métodos como si hubiera interpretado que lo que se pedía no era que generase el código, sino que generase un esqueleto sin la definición de los métodos. Si bien es cierto que este hecho puede estar relacionado con el contenido de los *prompts*, la falta de una validación con tests incrementa el número de veces que sucede esta situación y provoca que interprete erróneamente lo que ha solicitado el usuario.

Por lo tanto, a pesar de haber modificado el *prompt* múltiples veces, tal como se verá en el siguiente capítulo donde se comparan los resultados, esta metodología no ofrece la mejor respuesta. Además, dado que no se solicitan tests, tampoco se puede proporcionar más información a la persona que ha solicitado el código.

En cualquier caso, dado que es la metodología que menos texto y comprobaciones tiene que generar, también ofrece un tiempo de respuesta considerablemente bueno. Por este mismo motivo, es también la metodología que ha provocado menos errores relacionados con tiempos de espera.

4.6. Generación de código a partir de lenguaje natural y tests

A continuación, esta generación corresponde a la metodología número 2 que se ha detallado en el primer capítulo. Por lo tanto, es la que envía la colección de tests ya creados para que valide el código con ellos. Consecuentemente, se convierte automáticamente en la metodología que más *tokens* necesita y, además, la que requiere más trabajo para poder utilizarla, ya que se deben crear los tests manualmente antes de enviarlos.

Precisamente por este motivo, durante la generación de código con esta metodología es cuando más problemas han surgido referentes a tiempos de espera. Hasta el punto de que ha habido clases que no se han podido evaluar porque no llegaban a generar nunca una respuesta. Por ejemplo, con clases donde solo se deben generar un número muy reducido de métodos, como podría ser 4, no ha habido ningún problema. En cambio, cuando una clase requiere varios métodos más, es cuando se producían los problemas.

No obstante, esta metodología se ha podido aplicar en este caso porque se partió de un proyecto ya existente como se explicó en la introducción de este documento. Por lo tanto, se ha requerido una colección de tests para poder ser enviados.

Cabe destacar que el LLM no compila el código generado e intenta pasar los tests ejecutándolos, si no que, a partir del texto generado (el código), determina si un test puede pasar o no. Como resultado, existe el riesgo de que aparezcan falsos positivos o falsos negativos. Es decir, cuando se han analizado si han pasado los tests, o no, ha habido situaciones en las que el LLM indicaba que había un test que no lo había superado, pero realmente no era cierto. Por lo tanto, no se puede tener la certeza de que la información que da respecto a la superación de los tests es totalmente cierta.

Además, algunos de los motivos por los que se indica que un test ha fallado realmente no deberían suponer un problema. Esto se ha producido especialmente en la gestión de las excepciones. La colección de tests enviada está pensada para un código que ha utilizado unas excepciones en concreto, pero, como el LLM puede generar el código de una manera diferente, puede tratar una misma situación con una excepción diferente. Consecuentemente, el test falla, pero no es una situación en la que realmente se encuentre un error relevante.

Finalmente, se han observado situaciones que no deberían haberse producido con esta metodología. Tal como se ha dicho, se ha proporcionado una colección de tests con la cual se debe validar el código generado. Sin embargo, cuando el LLM devuelve la respuesta, se han encontrado varios intentos donde ha añadido más tests de los que se le han proporcionado. Es decir, no ha hecho caso a una parte del *prompt* donde se le especificaba que no debía generar tests adicionales, sino que simplemente tenía que evaluar los que ya tenía.

El archivo de la biblioteca que contiene los tests de la clase *QueueArrayImpl* tiene 5 tests. Este es el número que se ha repetido en más ocasiones entre los diversos intentos de generación de la misma clase. No obstante, como se ha dicho, se han encontrado varios casos donde se han añadido más tests de la cuenta. Además, el hecho de tener más tests de los que se han proporcionado no ha provocado que haya una mejora en el código.

```
<time value="0:00:52.457889"/>
<test id="1" result="OK" case="Verify add(E elem) method" expected="Method should add an item to the queue" result="Method added an item to the queue" reason=""/>
<test id="2" result="OK" case="Verify peek() method" expected="Method should retrieve the first item in the queue without deleting it" result="Method retrieved the first item in the queue without deleting it" reason=""/>
<test id="3" result="OK" case="Verify poll() method" expected="Method should retrieve and delete the first item in the queue" result="Method retrieved and deleted the first item in the queue" reason=""/>
<test id="4" result="OK" case="Verify isEmpty() method" expected="Method should return true if the queue is empty and false otherwise" result="Method returned true when called on an empty queue" reason=""/>
<test id="5" result="FAILED" case="Verify isFull() method" expected="Method should return true if the queue is full and false otherwise" result="Method returned false when called on a full queue" reason="The isFull() method is not correctly implemented"/>
<test id="6" result="OK" case="Verify size() method" expected="Method should return the number of items in the queue" result="Method returned the correct number of items in the queue" reason=""/>
<test id="7" result="OK" case="Verify values() method" expected="Method should return an iterator over the items in the queue" result="Method returned an iterator over the items in the queue" reason=""/>
```

Figura 15. Captura de pantalla de una colección de tests creada correspondiente al intento 8 de la cola. Fuente: original.

4.7. Generación de código y tests a partir de lenguaje natural

Pasando a la tercera de las metodologías explicadas al inicio del documento, se deben comentar varios aspectos. El primero de ellos es que, a pesar de no enviar una cantidad muy alta de *tokens*, ya que no contiene ninguna colección de tests, se producían tiempos de espera considerablemente altos. Consecuentemente, tal como se explicará más adelante, el tiempo de espera no solo está asociado a la cantidad de *tokens* enviados, sino también a lo que se pide. Así, durante la generación de código con esta metodología, se han sufrido problemas relacionados con los *timeouts*.

Aun así, el código generado que se ha obtenido tenía un comportamiento muy similar, en cuanto a la validez se refiere, al código original con el que se ha comparado (el de la biblioteca utilizada). Este resultado está altamente relacionado con el concepto de *chain-of-thought*. Debido a que el LLM es quien debe generar los tests para validar el código, se le dota con la capacidad de proporcionar código más preciso, ya que es capaz de descomponer el requisito solicitado por el usuario en aspectos más concretos.

Por ejemplo, como tiene que generar los tests para varios escenarios, el código generado tiene más probabilidades de contemplarlos. Es decir, si determina que debe haber un test que controle si un vector está vacío, el código contendrá el control de ese escenario.

Por otro lado, se ha observado que el número de tests siempre varía y raramente repite el mismo tipo de estos a pesar de utilizar exactamente el mismo *prompt* y asignar el atributo de la *temperature* a 0. A continuación se muestra un ejemplo de dos versiones de tests generados cuando se le solicitaba una lista enlazada:

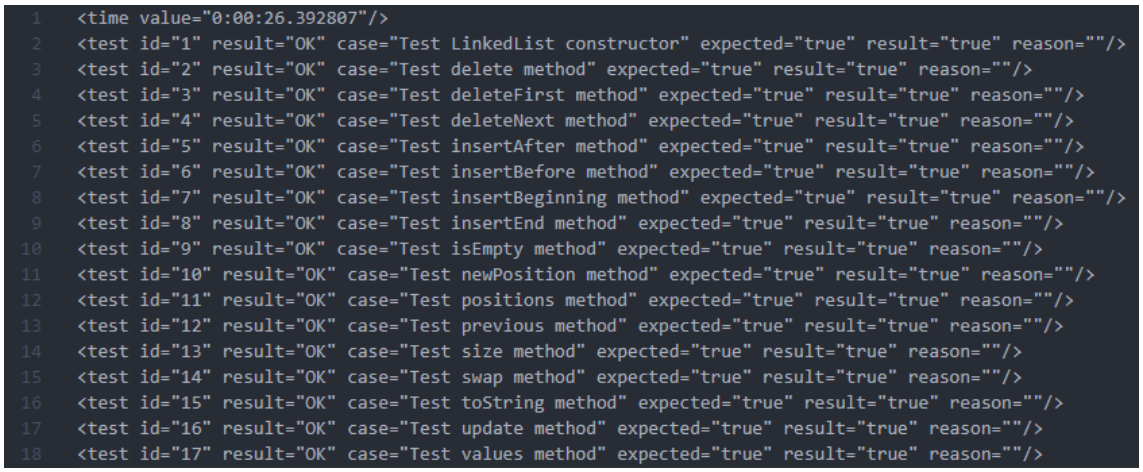


Figura 16. Captura de pantalla de una colección de tests creada correspondiente al intento 1 de la lista enlazada. Fuente: original.

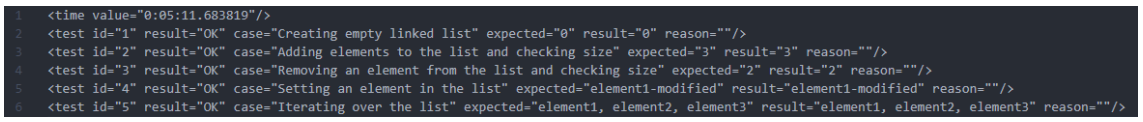


Figura 17. Captura de pantalla de una colección de tests creada correspondiente al intento 2 de la lista enlazada. Fuente: original.

Como se puede ver, la diferencia entre las dos llamadas es enorme. En la primera, se han generado hasta 17 tests que verifican el comportamiento del código generado. Mientras que en la segunda solo se han generado 5.

También es interesante señalar que el nombre o descripción que asigna a los tests nunca es el mismo. De nuevo, aparece el alto componente de aleatoriedad del LLM.

Además, si se analizan los archivos con el código generado, el archivo que va asociado con los primeros tests contiene todos los métodos sin cuerpo, mientras que el segundo, a pesar de haber creado menos tests para validar el comportamiento, sus métodos sí tienen cuerpo y, además, el comportamiento que tiene es correcto, aunque no sea el más eficiente posible.

```java

public class LinkedList<E> extends Object implements List<E> {

    //Empty list
    public LinkedList() {}

    //Delete received position
    public E delete(Position<E> node) {
        //code for deleting node at given position goes here
    }

    //Delete the first position in the list
    public E deleteFirst() {
        //code for deleting first node goes here
    }

    //Delete the next position
    public E deleteNext(Position<E> node) {
        //code for deleting node after given position goes here
    }

    //Add an item after the received position
    public Position<E> insertAfter(Position<E> node, E elem) {
        //code for inserting element after given position goes here
    }

    //Add an item before the received position
    public Position<E> insertBefore(Position<E> node, E elem) {
        //code for inserting element before given position goes here
    }
```

Figura 18. Captura de pantalla del código generado correspondiente al intento 1 de la lista enlazada. Fuente: original.

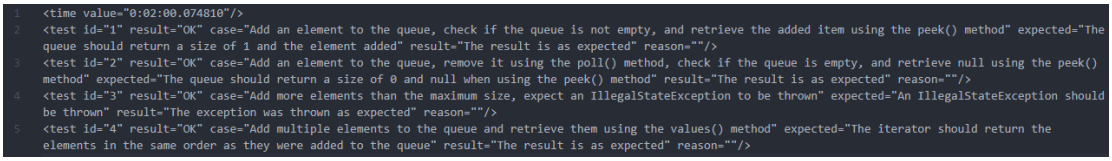
```java
    public LinkedList() {
        header = new LinkedNode<>(null, null, null);
        trailer = new LinkedNode<>(header, null, null);
        header.setNext(trailer);
    }

    protected LinkedNode<E> checkPosition(Position<E> p)
            throws IllegalArgumentException {
        if (p == null || !(p instanceof LinkedNode)) {
            throw new IllegalArgumentException("Invalid position");
        }
        LinkedNode<E> node = (LinkedNode<E>) p;
        if (node.getNext() == null) {
            throw new IllegalArgumentException(
            "Position is not in the list anymore");
        }
        return node;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public Position<E> first() throws EmptyListException {
        if (isEmpty()) {
            throw new EmptyListException("List is empty");
        }
        return header.getNext();
    }
```

Figura 19. Captura de pantalla del código generado correspondiente al intento 2 de la lista enlazada. Fuente: original.

Por lo tanto, el hecho de tener una colección de tests más grande que otra no garantiza que el código sea de mejor calidad. Ahora bien, es necesario dejar constancia que este hecho mencionado ha sucedido en una clase que contiene bastantes métodos y que durante la generación del código hubo varios

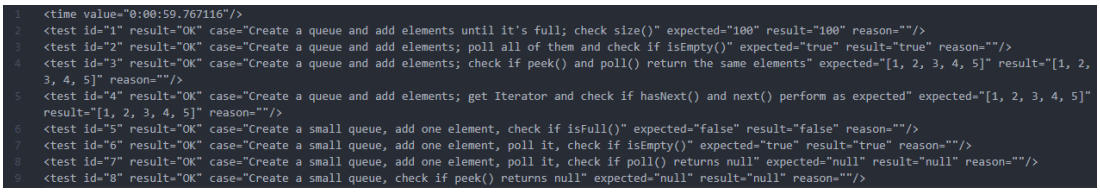
problemas referentes a los tiempos de respuesta. De hecho, en la Figura 16 se observa que el tiempo utilizado para generar la clase fue de más de 5 minutos, ya que entró en bucle con varios *timeouts*.

De esta manera, es necesario analizar otras clases con menos métodos para terminar de verificar si la cantidad de texto que tiene que generar influye en la calidad del código generado. Para hacerlo, se mostrará la colección de tests creada para una cola de dos intentos diferentes.



```
<time value="0:02:00.074810"/>
<test id="1" result="OK" case="Add an element to the queue, check if the queue is not empty, and retrieve the added item using the peek() method" expected="The queue should return a size of 1 and the element added" result="The result is as expected" reason=""/>
<test id="2" result="OK" case="Add an element to the queue, remove it using the poll() method, check if the queue is empty, and retrieve null using the peek() method" expected="The queue should return a size of 0 and null when using the peek() method" result="The result is as expected" reason=""/>
<test id="3" result="OK" case="Add more elements than the maximum size, expect an IllegalStateException to be thrown" expected="An IllegalStateException should be thrown" result="The exception was thrown as expected" reason=""/>
<test id="4" result="OK" case="Add multiple elements to the queue and retrieve them using the values() method" expected="The iterator should return the elements in the same order as they were added to the queue" result="The result is as expected" reason=""/>
```

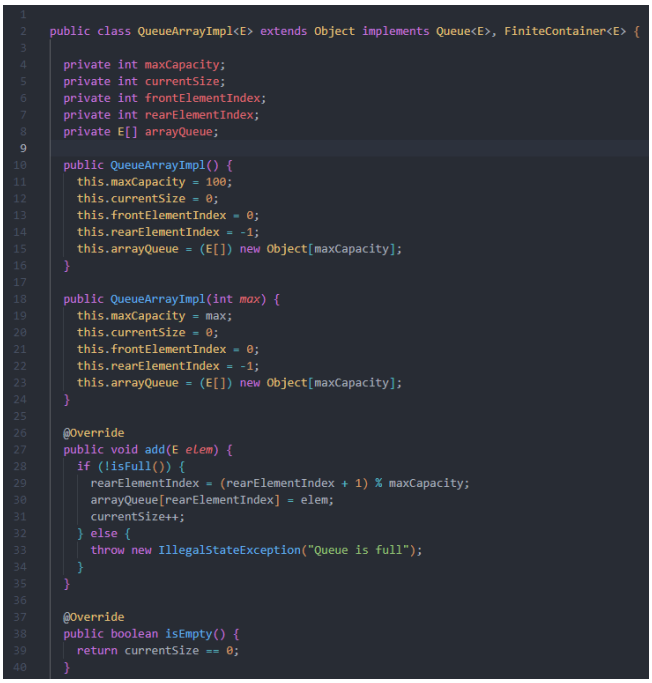

Figura 20. Captura de pantalla de una colección de tests creada correspondiente al intento 13 de la cola. Fuente: original.



```
<time value="0:00:59.767116"/>
<test id="1" result="OK" case="Create a queue and add elements until it's full; check size()" expected="100" result="100" reason=""/>
<test id="2" result="OK" case="Create a queue and add elements; poll all of them and check if isEmpty()" expected="true" result="true" reason=""/>
<test id="3" result="OK" case="Create a queue and add elements; check if peek() and poll() return the same elements" expected="[1, 2, 3, 4, 5]" result="[1, 2, 3, 4, 5]" reason=""/>
<test id="4" result="OK" case="Create a queue and add elements; get Iterator and check if hasNext() and next() perform as expected" expected="[1, 2, 3, 4, 5]" result="[1, 2, 3, 4, 5]" reason=""/>
<test id="5" result="OK" case="Create a small queue, add one element, check if isFull()" expected="false" result="false" reason=""/>
<test id="6" result="OK" case="Create a small queue, add one element, poll it, check if isEmpty()" expected="true" result="true" reason=""/>
<test id="7" result="OK" case="Create a small queue, add one element, poll it, check if poll() returns null" expected="null" result="null" reason=""/>
<test id="8" result="OK" case="Create a small queue, check if peek() returns null" expected="null" result="null" reason=""/>
```


Figura 21. Captura de pantalla de una colección de tests creada correspondiente al intento 15 de la cola. Fuente: original.

Si se analiza el código, si bien es cierto que no tienen exactamente el mismo comportamiento, no existe la diferencia que se ha producido con la lista enlazada.



```
public class QueueArrayImpl<E> extends Object implements Queue<E>, FiniteContainer<E> {

  private int maxCapacity;
  private int currentSize;
  private int frontElementIndex;
  private int rearElementIndex;
  private E[] arrayQueue;

  public QueueArrayImpl() {
    this.maxCapacity = 100;
    this.currentSize = 0;
    this.frontElementIndex = 0;
    this.rearElementIndex = -1;
    this.arrayQueue = (E[]) new Object[maxCapacity];
  }

  public QueueArrayImpl(int max) {
    this.maxCapacity = max;
    this.currentSize = 0;
    this.frontElementIndex = 0;
    this.rearElementIndex = -1;
    this.arrayQueue = (E[]) new Object[maxCapacity];
  }

  @Override
  public void add(E elem) {
    if (!isFull()) {
      rearElementIndex = (rearElementIndex + 1) % maxCapacity;
      arrayQueue[rearElementIndex] = elem;
      currentSize++;
    } else {
      throw new IllegalStateException("Queue is full");
    }
  }

  @Override
  public boolean isEmpty() {
    return currentSize == 0;
  }
```


Figura 22. Captura de pantalla del código generado correspondiente al intento 13 de la cola. Fuente: original.

```
public class QueueArrayImpl<E> extends Object implements Queue<E>, FiniteContainer<E> {

    private E[] elements;
    private int size;
    private int head = 0; //index of the first element
    private int tail = -1; //index of the last element

    public QueueArrayImpl() {
        this(100);
    }

    public QueueArrayImpl(int max) {
        elements = (E[]) new Object[max];
    }

    @Override
    public void add(E elem) {
        if(isFull()) throw new IllegalStateException("Container is full.");
        tail = (tail + 1) % elements.length;
        elements[tail] = elem;
        size++;
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }

    @Override
    public boolean isFull() {
        return size == elements.length;
    }

    @Override
    public E peek() {
        if(isEmpty()) return null;
        return elements[head];
    }
```

Figura 23. Captura de pantalla del código generado correspondiente al intento 15 de la cola. Fuente: original.

4.8. Generación de código aplicando cadenas de *LangChain*

En cuanto a esta metodología, se ha preparado el prototipo para poder utilizar una cadena con el objetivo de enviar más de un *prompt* al LLM. Concretamente, el primer *prompt* se ha utilizado para enviar un ejemplo sobre qué tiene que generar y, en el segundo, solicitarle el código aplicando una de las tres metodologías descritas en los apartados anteriores.

Después de llevar a cabo varias pruebas, se ha tenido que descartar esta metodología debido a los altos tiempos de espera que se producían e, incluso, terminaban produciendo un *timeout* eliminando cualquier posibilidad de recibir la respuesta generada. Adicionalmente, esta metodología requiere que exista un ejemplo para el código que se quiere crear. En consecuencia, se añade una dificultad como pasa en la metodología donde se envía una colección de tests ya creados.

Por lo tanto, ha sido una vía que no se ha podido explorar y no se tendrá en cuenta en el capítulo de análisis. Aún así, es importante destacar que, cuando el LLM sea capaz de ofrecer una eficiencia aceptable con esta metodología, será interesante evaluar los resultados.

4.9. Generación de código aplicando *active prompting*

Finalmente, esta ha sido la última metodología que se ha puesto a prueba. Concretamente, como ya se ha explicado en el capítulo 3 de esta memoria, se puede aplicar mezclando otras metodologías. Por lo tanto, los tiempos de espera de esta metodología dependen directamente de las otras (si no se quiere validar el código con tests, con tests propios o creados por él mismo).

Por lo tanto, aunque no se pueda hacer con una cadena como estaba planificado inicialmente, se ha planteado de manera que se vuelve a enviar el código generado con los comentarios que haya hecho la persona que está generando el código. De esta manera, se evitan los problemas asociados a la poca eficiencia de la cadena y se modifica el código generado inicialmente.

Así, la mejora del nuevo código generado vuelve a depender de un factor, que es la calidad de la información que le ha proporcionado la persona que está solicitando el código. De hecho, si la información proporcionada es muy concreta y abastece todo lo que se debe modificar, con una sola llamada más al LLM es capaz de modificar el código y cumplir todos los requisitos de este.

El procedimiento que se debe seguir para aprovechar al máximo esta metodología consiste en evaluar la respuesta que se ha obtenido inicialmente. En caso de que haya algún aspecto que se deba modificar o replantear, siempre se debe indicar utilizando órdenes. Es decir, el LLM utilizado entiende y mejora las indicaciones si estas son dadas como órdenes. Y, en caso de ser varias, que se den en diferentes oraciones como si se tratara de una lista. De hecho, este tipo de comportamiento ya se explicó en la sección donde se hablaba del diseño de los *prompts*. Teniendo en cuenta que las indicaciones que se deben dar se enviarán en un *prompt*, es coherente que también se tengan que enviar siguiendo el mismo formato cuando se aplica *active prompting*. Por lo tanto, se deben evitar oraciones largas y poco claras y expresiones como: *debería hacer*, *quizás estaría mejor que* o *creo que*.

Un ejemplo de mensaje que cumple con los requisitos que se han mencionado podría ser el siguiente:

> *Debes añadir un atributo que permita contar el número de elementos de la tabla. Debes eliminar todos los comentarios del código. Modifica el nombre del método calculateNumber por getSize.*

Como se puede ver, las oraciones son cortas, claras e indican órdenes que debe seguir.

Finalmente, se ha detectado que, si se lleva a cabo este proceso muchas veces sobre un mismo código, los resultados que devuelve empiezan a olvidar parte de lo que se había dicho inicialmente. En consecuencia, no es recomendable iterar muchas veces esta metodología sobre un mismo código. En caso de que no devuelva el código esperado, se obtiene un mejor resultado si se vuelve a plantear el *prompt* inicial. De hecho, esta peculiaridad se da porque el LLM utilizado no tiene memoria si se supera la limitación de *tokens*. Por eso, no puede tener en cuenta lo que se le ha dicho inicialmente.

Cuando se llevaron a cabo las llamadas a la API de este proyecto, la limitación de *tokens* estaba en 4.096, mientras que en el momento de entregar esta memoria se ha aumentado a más de 16.000. Con esta mejora es probable que se puedan llevar a cabo más iteraciones sobre esta metodología, pero, una vez

supere la limitación, empezará a ocurrir lo que ha sucedido durante este estudio.

4.10. Errores y problemas detectados durante la generación de código

Para finalizar el capítulo, se comentarán varios errores o comportamientos no deseados que se han visto generalizados, independientemente de la metodología empleada.

El primero de los errores que se observó y que se ha mantenido hasta el final del trabajo, a pesar de haber aplicado medidas de mitigación, son los *timeouts*. Al principio, debido a la complejidad que tenía el *prompt*, ocurría siempre. Por lo tanto, tras cambiarlos, aplicando la técnica que se ha descrito en los apartados anteriores, se logró disminuir el número de veces que aparecían. No obstante, se han detectado comportamientos que se han repetido durante toda la ejecución de las pruebas.

Como ya se mencionó en el capítulo 3, el prototipo dispone de una manera de ejecutarse que le permite generar múltiples veces una misma clase. Inicialmente, esta opción no se podía utilizar, ya que se detectó que la API del LLM utilizado detecta si se llevan a cabo muchas consultas consecutivas. Por lo tanto, después de generar un par de clases, el prototipo ya no generaba más porque siempre retornaba un *timeout*.

Con el objetivo de eludir esta política de seguridad contra el *spam*, se tuvo que implementar un tiempo de espera aleatorio entre cada una de las llamadas. Gracias a esto, se logró mitigar considerablemente el problema de los timeouts.

Aún así, había clases que todavía continuaban produciendo esta situación. Concretamente, se observó que las clases que tenían más métodos tendían más a provocar un error por tiempo de espera. Consecuentemente, ha habido varias clases de la biblioteca que no se han podido poner a prueba por este motivo.

Adicionalmente, en aquellas clases con bastantes métodos donde sí se podía ejecutar el prototipo, a pesar de tener tiempos de espera considerablemente altos, se generaba el código, pero la mayoría de las veces lo hacía con métodos sin cuerpo. Concretamente, cuando lo hacía, siempre dejaba un comentario dentro del método, como se puede ver en la Figura 18.

Incluso, ha habido clases donde en ninguna ocasión ha sido capaz de crear métodos con cuerpo a pesar de no haber dado excesivos problemas al momento de generar el código. Consecuentemente, la falta de cuerpo en estos métodos no se debe al hecho de que tenga un contexto con muchos *tokens*, sino que se debe a la falta de comentarios dentro del archivo original desde donde se obtiene el contexto.

```
23      * Attribute that determines compatibility between objects
24      * serializable of the same class. It is calculated
25      * using a method of the Utilities class.
26      */
27     private static final long serialVersionUID = Utils.getSerialVersionUID();
28
29
30     public DirectedGraphImpl() {
31         super();
32     }
33
34
35     protected VertexImpl<E, L> createVertex(E value) {
36         return new DirectedVertexImpl<>(value);
37     }
38
39
40     public DirectedEdge<L, E> newEdge(Vertex<E> src,
41                                       Vertex<E> dest) {
42         DirectedVertexImpl<E, L> s = (DirectedVertexImpl<E, L>) src;
43         DirectedVertexImpl<E, L> d = (DirectedVertexImpl<E, L>) dest;
44         DirectedEdgeImpl<L, E> edge = new DirectedEdgeImpl<>(s, d);
45         edge.add2Graph(this);
46         return edge;
47     }
48
49
50     public DirectedEdge<L, E> getEdge(Vertex<E> src,
51                                       Vertex<E> dest) {
52         DirectedEdge<L, E> res = null;
53         Iterator<Edge<L, E>> edges = edgesWithSource(src);
54         while (edges.hasNext() && res == null) {
55             DirectedEdge<L, E> edge = (DirectedEdge<L, E>) edges.next();
56             if (edge.getVertexDst() == dest)
57                 res = edge;
58         }
59         return res;
60     }
61
```

Figura 24. Captura de pantalla del código fuente original de la biblioteca de la asignatura Diseño de Estructura de Datos del archivo *DirectedGraphImpl.java*. Fuente: original.

Como se puede observar en la Figura 24, los métodos que tiene que generar no tienen documentación. Por lo tanto, de este caso se puede extraer que el LLM no es capaz de generar el cuerpo del método con solo el nombre de este. Inicialmente, se puede pensar que el nombre del método y los parámetros que recibe pueden llegar a ser suficientes para dar contexto al LLM. Con este caso, queda demostrado que no siempre es posible.

De hecho, es importante destacar que no siempre es posible, dejando espacio a situaciones donde sí que lo es porque hay escenarios donde se ha observado que es capaz de crear el cuerpo del método. Por ejemplo, la clase *MergeSort* no contiene comentarios asociados al método. A pesar de eso, hay situaciones donde sí que ha sido capaz de crear el código.

Consecuentemente, la cabecera del método se convierte en un contexto insuficiente para garantizar que el código se generará correctamente.

Otro aspecto relevante que destacar es que, como se han realizado múltiples llamadas a la API durante todo el semestre, se detectó que el cambio de versión del día 3 de mayo provocó que los resultados empeoraran drásticamente con el *prompt* que se estaba utilizando anteriormente. De hecho,

el cambio fue tal que pasó de generar código correcto en un número reducido de veces a directamente generar respuestas como la siguiente.

Figura 25. Respuesta obtenida al ejecutar el prototipo justo después de un cambio de versión con el prompt funcional anterior. Fuente: original.

En consecuencia, se ha detectado que se pueden producir diferentes resultados utilizando exactamente el mismo *prompt* en diferentes versiones. Además, el cambio es muy drástico, ya que se pasó de generar siempre código (aunque muchas veces lo generaba con métodos sin cuerpo) a generar respuestas que directamente no contenían código. De hecho, el texto *(code here)* es una parte del *prompt* que se envía al requerir el código al LLM.

Continuando con los errores que se han podido detectar, se ha observado que hay distintas franjas horarias donde se generaban más *timeouts* que otras. Por lo tanto, parece evidente que, cuando el modelo está más monopolizado, es más difícil poder generar el código y, en general, obtener respuesta con la API. Así, se tuvieron que ir haciendo pruebas en distintas franjas horarias para poder determinar la que mejor rendimiento ofrecía.

Después de varios días, se pudo concluir que la mejor franja horaria para lanzar las llamadas a la API es entre las 2 AM y las 6 AM. Es decir, la franja horaria que coincide cuando es de noche en Europa y cuando la jornada laboral, en principio, debería haber terminado en América. Además, corroborando este hecho, también se ha detectado que durante los días que caen en fines de semana se reduce el número de *timeouts*.

Ahora bien, esta conclusión es un arma de doble filo, ya que si otras personas usuarias del LLM detectan esta situación, podrán automatizar las llamadas que tengan que hacer para hacerlas durante las horas mencionadas. Así, aunque esta afirmación se ha podido hacer durante los primeros meses de vida de la versión del LLM utilizado, en el futuro puede cambiar drásticamente.

El siguiente punto a tener en cuenta, aunque no sea un error como tal, es un comportamiento extraño que debe ser reportado. Concretamente, ha habido métodos donde el LLM ha añadido etiquetas como *@Override* a pesar de no necesitarlo. Además, en algunas ocasiones ha entendido que era necesario heredar clases que no lo eran. Por ejemplo, heredar la clase *Object* en una clase de Java.

En cuanto a los tests generados, se han encontrado varias situaciones en las que estos muestran una realidad que es imposible que sea cierta. Concretamente, existen tests que son falsos negativos, donde se indica que un test ha fallado, pero realmente lo ha hecho por falta de contexto. Es decir, es un error que no debería ser relevante al tratar con el código generado. Por otro lado, hay situaciones donde se muestra que todos los tests han pasado correctamente. En cambio, el cuerpo de los métodos está vacío. Por lo tanto, los tests están mintiendo sobre el correcto funcionamiento del código.

Este hecho se ha podido detectar con la clase *LinkedList* en el intento número 9.

```
<time value="0:04:42.783682"/>
<test id="1" result="OK" case="Creating an instance of an empty linked list" expected="true" result="(new LinkedList<Integer>()).isEmpty()" reason=""/>
<test id="2" result="OK" case="Adding a single element to the beginning of a linked list" expected="1" result="new LinkedList<Integer>().insertBeginning(1).size()" reason=""/>
<test id="3" result="OK" case="Adding two elements to a linked list, one to the beginning, and one to the end" expected="2" result="new LinkedList<Integer>().insertBeginning(1).insertEnd(2).size()" reason=""/>
<test id="4" result="OK" case="Deleting the first element of a linked list" expected="1" result="new LinkedList<Integer>().insertBeginning(1).insertEnd(2).deleteFirst().size()" reason=""/>
<test id="5" result="OK" case="Updating an element of a linked list" expected="2" result="new LinkedList<Integer>().insertBeginning(1).insertEnd(2).update(new LinkedList<Integer>().insertBeginning(1).positions().next(), 2).values().next()" reason=""/>
```

Figura 26. Captura de pantalla donde se indica que todas los tests generados pasan correctamente. Fuente: original.

En cambio, si se revisa el código generado, se confirma que es completamente imposible que los tests se hayan superado correctamente.

```
public class LinkedList<E> extends Object implements List<E> {

    protected static class LinkedNode<E> implements Position<E> {
        /* attributes */
    }

    public LinkedList() {
        /* constructor body */
    }

    public E delete(Position<E> node) {
        /* method body */
    }

    public E deleteFirst() {
        /* method body */
    }

    public E deleteNext(Position<E> node) {
        /* method body */
    }

    public Position<E> insertAfter(Position<E> node, E elem) {
        /* method body */
    }

    public Position<E> insertBefore(Position<E> node, E elem) {
        /* method body */
    }
```

Figura 27. Captura de pantalla del inicio del código generado de la clase *LinkedList* sin cuerpo en los métodos. Fuente: original.

En consecuencia, puede haber situaciones donde los tests tengan un resultado que no es posible. Estos casos se han observado especialmente en aquellas clases que envían un contexto muy grande al LLM.

Finalmente, otro problema que se ha detectado, que ya se introdujo anteriormente, es que cuanto más largo es un contexto que se envía, más probabilidad hay de que no siga alguna de las reglas dictadas en el *prompt*. Dado que el prototipo se basa en ofrecer una respuesta en formato XML, cuando esta no mantiene ese formato muestra un error. Y, concretamente, este

error solo aparecía cuando la cantidad de métodos que se tenían que generar y el contexto que se enviaba era considerablemente más largo que otros.

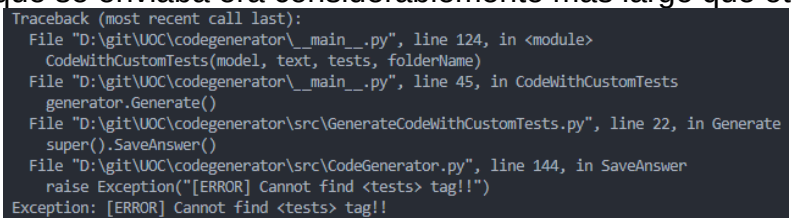


Figura 28. Error que ilustra la situación que se daba cuando la respuesta no seguía el formato solicitado. Fuente: original.

5. Análisis comparativo

5.1. Análisis por variables y metodología

El análisis que se ha llevado a cabo en este capítulo no contiene todas las gráficas obtenidas. Estas han sido tratadas y se han seleccionado solo aquellas que son más significativas para analizar y comparar los resultados. No obstante, si el lector de este documento desea consultarlas, puede acceder al repositorio de gráficas mediante el Anexo VI. Adicionalmente, si también se desean consultar todos los resultados obtenidos directamente de las llamadas al LLM, se pueden revisar dentro de la carpeta *data* del repositorio del código en el Anexo IV.

Por lo tanto, en este capítulo se llevará a cabo un análisis entre las diferentes metodologías que se han aplicado respecto a varias variables.

En primer lugar, se realizará un análisis sobre la calidad del código generado, es decir, lo que se ha denominado como precisión. Solo se tratarán las tres metodologías que se han automatizado, es decir, la generación de código sin validación con tests, la que valida el código con tests proporcionados en el mismo *prompt* y, finalmente, la que valida el código generado con una colección de tests propia.

En segundo lugar, se comparará cuál es el grado de mejora cada vez que se itera el código. Para simular una situación real, se entenderá como iteración cada vez que un usuario interactúa con el LLM proporcionando información respecto al código previamente generado.

Finalmente, es necesario llevar a cabo un estudio sobre la eficiencia temporal que tiene cada una de estas metodologías, ya que es importante analizar si son viables.

5.1.1. Precisión

En este apartado, se llevará a cabo un análisis comparativo entre las tres metodologías empleadas según la calidad de su código. Dado que no se pueden presentar todas las gráficas de todas las clases generadas, se han seleccionado aquellas que se consideran más relevantes para constatar ciertos aspectos que ya se han introducido anteriormente. Es decir, se han seleccionado las gráficas más significativas y representativas.

Además, como se ha mencionado en el capítulo anterior, ha habido clases que directamente no han generado ningún método con cuerpo. En consecuencia, no se puede llevar a cabo un análisis comparativo sobre estas clases porque no han generado un código válido en ningún caso.

Antes, pero, se debe definir en función de qué variables se determinará la precisión del código . Concretamente, se establecen 5 categorías posibles:

- Sin código: representa esos casos donde se han generado los métodos, pero estos están sin cuerpo.
- Incorrecto: esta categoría representa los intentos donde el código generado, aunque es sintácticamente correcto, no ofrece el comportamiento deseado.
- No eficiente: representa aquellos casos en los que le código permite ofrecer un funcionamiento correcto, pero de manera no eficiente.
- Correcto: muestra aquellos casos en los que el código generado ofrece el comportamiento solicitado y es eficiente, pero no tiene en cuenta el contexto de la clase y define algunos componentes que no son necesarios, ya que vienen dados por las clases padres.
- Perfecto: son aquellos casos que generan un código que cumple con las condiciones para ser considerado correcto y, además, tiene en cuenta el contexto de la clase y utiliza métodos o clases que vienen heredadas.

La primera de las clases seleccionadas, con la que se ha estado trabajando todo el semestre, es la llamada *QueueArrayImpl*. Esta clase es especialmente interesante porque permite categorizar perfectamente los códigos generados según las categorías anteriores. Concretamente, la versión más eficiente de esta clase es la que utiliza un vector donde almacena los elementos como si fuera un anillo. Por lo tanto, permite cuantificar cuál es el porcentaje de clases que se han podido generar teniendo en cuenta esta optimización.

Adicionalmente, esta clase no contiene un gran número de métodos a generar. Así, se utilizará como clase representativa de aquellas que tienen un contexto más corto.

Finalmente, con el objetivo de poder analizar las diversas metodologías entre las diferentes versiones que ha habido durante los últimos meses, se mostrarán las gráficas asociadas a la clase mencionada para la versión del día 3 de mayo y, posteriormente, para la versión del día 24 de mayo.

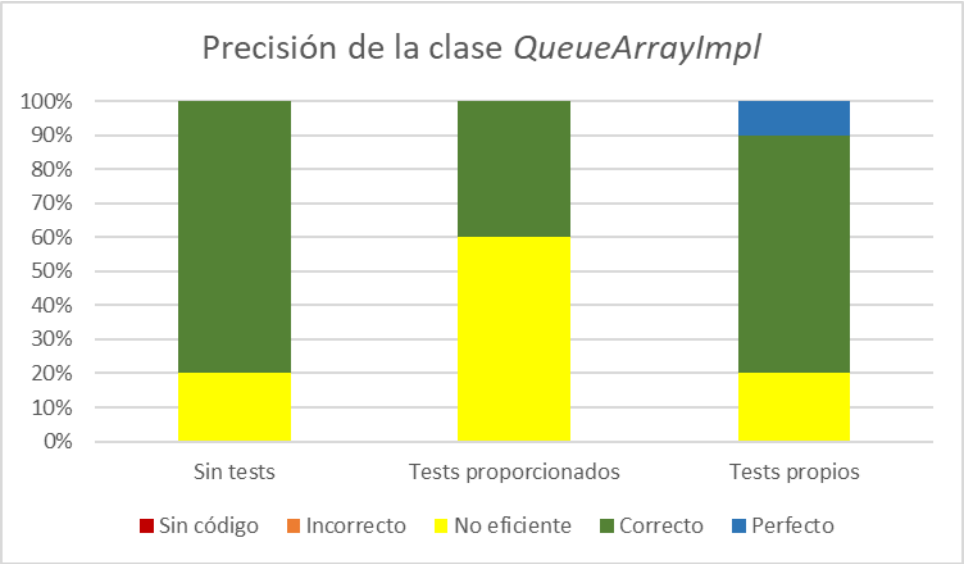


Figura 29. Gráfica que representa la precisión del código generado de la clase *QueueArrayImpl* con la versión del día 3 de mayo. Fuente: original.

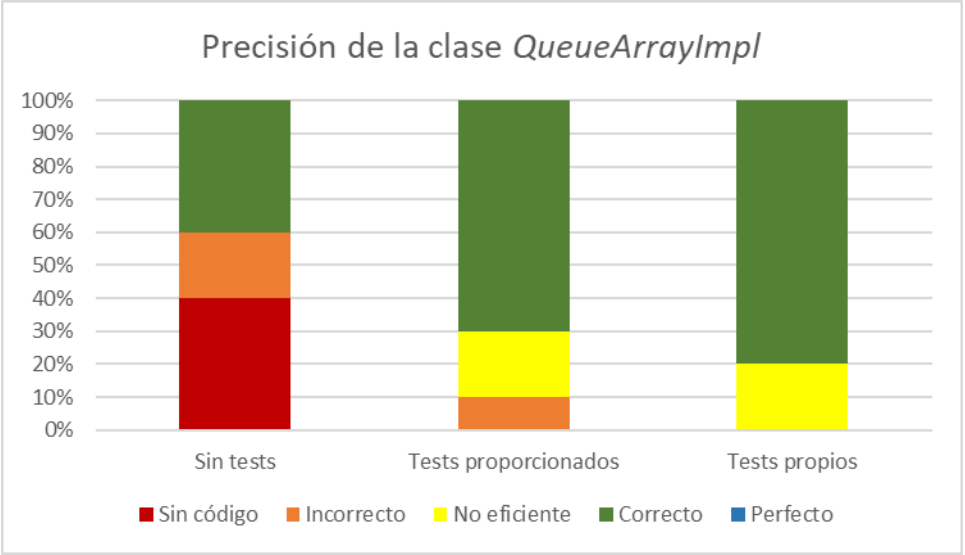


Figura 30. Gráfica que representa la precisión del código generado de la clase *QueueArrayImpl* con la versión del día 24 de mayo. Fuente: original.

Como se puede apreciar a simple vista, ya ha habido un cambio considerable entre las dos versiones con las que se ha ejecutado el código usando exactamente el mismo *prompt* para ambas. Aunque es cierto que el número de intentos no es considerablemente alto, 10 intentos por metodología y versión del LLM (un total de 60 archivos analizados), ya se pueden encontrar diferencias notables.

Por ejemplo, con la versión del día 3, los resultados obtenidos cuando no se validaba el código con tests no generaba nunca métodos sin cuerpo. Con el cambio de versión, se ha observado que esta metodología ha empeorado notablemente.

Sin embargo, sí que se observan patrones que se mantienen entre las dos gráficas, es decir, hay similitudes en cuanto a las metodologías empleadas entre diverses versiones. Concretamente, en ambas gráficas se mantiene que la que ofrece mejores resultados es la que genera los tests por sí misma. Este hecho ha sido comentado a lo largo de toda la memoria y con estas gráficas se puede constatar que es cierto.

Consecuentemente, la metodología que no utiliza tests como validación no parece una buena elección teniendo en cuenta que los resultados son muy inestables. Además, viendo los resultados y teniendo en cuenta que la metodología donde se proporciona una colección de tests requiere que se desarrollen manualmente, también se concluye que la mejor metodología, según las gráficas, es la que genera los tests por sí misma, ya que permite obtener mejores resultados sin tener que generar los tests previamente.

Ahora bien, es necesario llevar a cabo un estudio sobre cuál ha sido el comportamiento de los tests. Concretamente, se analiza si los tests se han superado correctamente, si hay alguno que está marcado como fallido, pero no es un error relevante o si ha fallado porque realmente el código es incorrecto. Así, las siguientes gráficas muestran los resultados de los tests para las dos metodologías que los utilizan.

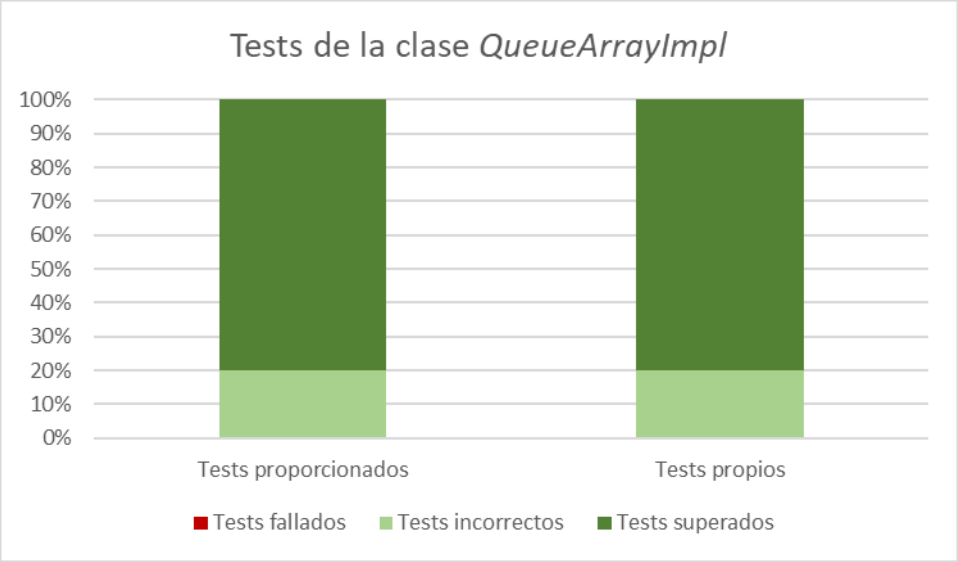


Figura 31. Gráfica que representa la corrección de los tests de la clase *QueueArrayImpl* con la versión del día 3 de mayo. Fuente: original.

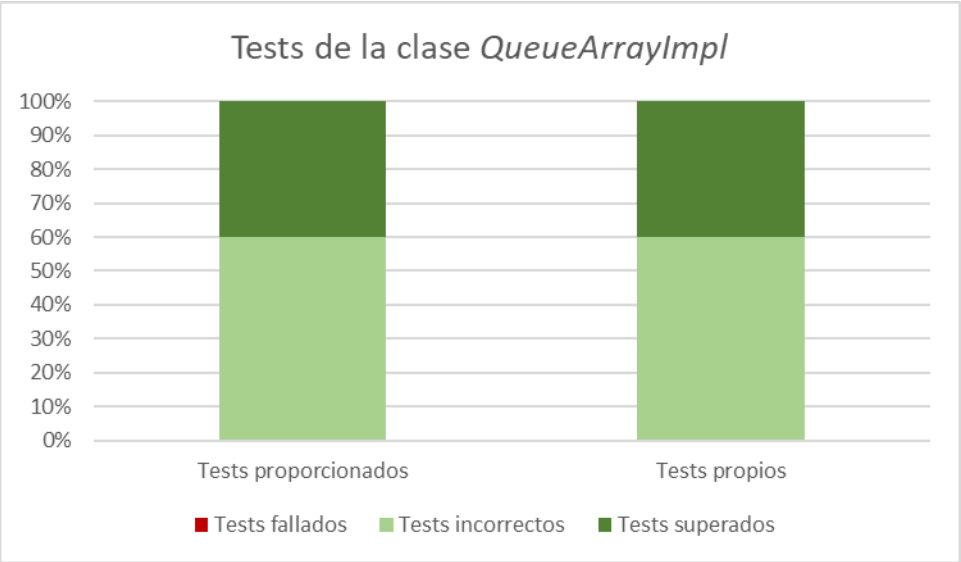


Figura 32. Gráfica que representa la corrección de los tests de la clase *QueueArrayImpl* con la versión del día 24 de mayo. Fuente: original.

Continuando con la tendencia de las gráficas anteriores, se observa que hay un empeoramiento entre las dos versiones en cuanto a la calidad de los tests, ya que han aumentado los casos en que fallan tests, pero realmente no es correcto. Teniendo en cuenta que se han analizado 10 casos para cada metodología, este número puede tener cierto sesgo porque la población usada no es muy alta. Lo que sí se puede afirmar es que la proporción entre las dos metodologías sigue siendo la misma.

En consecuencia, en lo que respecta a aquellas clases que no tienen un contexto muy grande a enviar, se concluye que la mejor metodología que se puede aplicar es la que genera los tests por sí misma.

En cambio, si se analizan aquellas clases que tienen un contexto mucho más grande, los resultados cambian drásticamente. Para analizar estos resultados, se utilizará la clase *LinkedList* como clase representativa, ya que contiene un gran número de métodos a desarrollar.

A continuación, sin abandonar el mismo modelo gráficas que se usaron en la clase anterior, la gráfica obtenida es la siguiente:

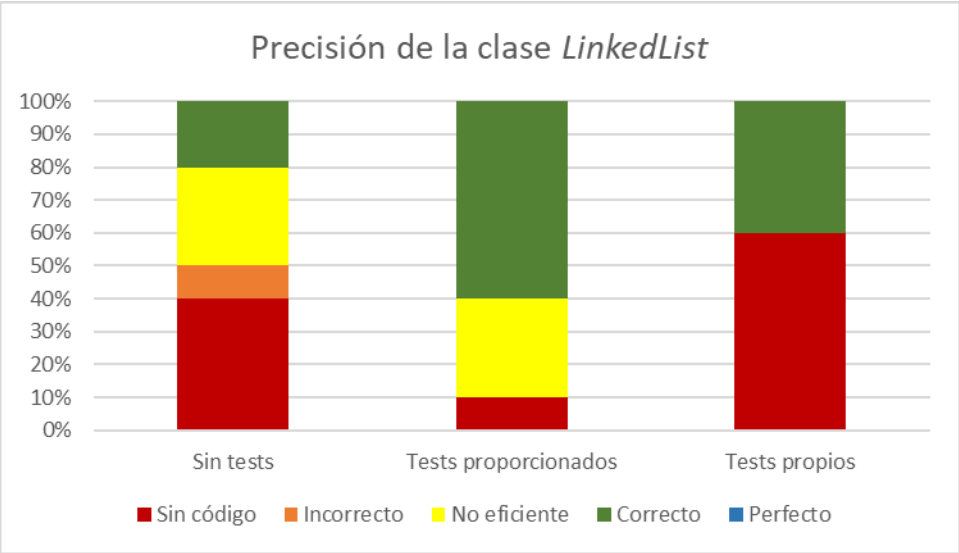


Figura 33. Gráfica que representa la corrección de los tests de la clase *LinkedList* con la versión del día 24 de mayo. Fuente: original.

Como se puede apreciar, los resultados empeoran considerablemente. En líneas generales, destaca especialmente que han aumentado los casos donde no hay código. Por lo tanto, se confirma lo que se ha ido diciendo a lo largo de toda la memoria y se llega a la conclusión de que, cuando el contexto es considerablemente largo, los resultados empeoran.

Ahora bien, en los 10 casos analizados, llama la atención que la metodología que usa tests propios tiene muchos casos donde no hay cuerpo en los métodos. Sin embargo, cuando el cuerpo sí que está presente, la solución que propone es correcta. Por lo tanto, esta metodología permite confirmar que, efectivamente, los resultados no son los esperados debido a la gran cantidad de *tokens* que se están enviando, lo que provoca que no se genera el código dentro de los métodos. Si el problema fuera la metodología, en los casos en los que se hubiera generado código, se mostrarían escenarios donde el código generado no es eficiente o directamente es erróneo.

Así, pues, estas son las gráficas que mejor representan los dos casos citados: cuando el contexto no es extremadamente largo o cuando sí lo es.

5.1.2. Escalabilidad de precisión por iteración

Tal como se comentó anteriormente en el apartado 4.9 de esta memoria, la mejora de la precisión del código generado depende totalmente de la información que el usuario proporcione en ese momento en función del resultado obtenido. Consecuentemente, resulta imposible ofrecer gráficas con datos útiles, ya que depende de la habilidad de la persona que interactúa con el LLM.

Adicionalmente, después de diversas pruebas, se ha podido observar que la mejora es tal que, incluso partiendo de un código con los métodos sin cuerpo, se puede llegar a una situación donde el código sea perfecto. Ahora

bien, cuanto peor esté el código generado en una primera instancia, más indicaciones son necesarias.

Para terminar, teniendo en cuenta que el LLM no tiene una capacidad de análisis lógico, si el usuario es capaz de proporcionarle ese punto de vista, el código generado realmente aplica los cambios solicitados correctamente, incluso aquellos que tienen que ver con planteamientos lógicos. Como resultado, se vuelve a uno de los puntos mencionados en la introducción, donde se afirmaba que la generación de código utilizando LLMs no puede reemplazar en ningún caso a la persona que programa, ya que esta debe tener un conocimiento para poder darle indicaciones al LLM. En pocas palabras, el LLM permite agilizar considerablemente el desarrollo de software, pero no sustituir la figura del programador o programadora.

Como ejemplo, reutilizando el caso del anillo que debe utilizar una cola para estar desarrollada eficientemente, se muestra el resultado de una primera llamada donde no trata la cola de manera correcta y, con la interacción de una persona desarrolladora, se logra aplicar correctamente un planteamiento lógico.

```
25      public void add(E elem) {
26          if (isFull()) throw new QueueFullException("Queue is full");
27          if (front == -1) ++front;
28          dataArray[++rear] = elem;
29          ++currentSize;
30      }
```

Figura 34. Captura de pantalla del método para añadir un elemento a una cola de manera ineficiente. Fuente: original.

Como el planteamiento que ha utilizado no es correcto, se le indica que debe cambiarlo con la siguiente frase.

You have to treat the dataArray vector as if it were a ring.

Una vez enviada la llamada al LLM, el código resultado se convierte en correcto y eficiente.

```
25      public void add(E elem) {
26          if (isFull()) throw new QueueFullException("Queue is full");
27          if (isEmpty()) {
28              front = rear = 0;
29          } else {
30              rear = (rear + 1) % maxSize;
31          }
32          dataArray[rear] = elem;
33          ++currentSize;
34      }
```

Figura 35. Captura de pantalla del método para añadir un elemento a una cola de manera eficiente. Fuente: original.

En resumen, queda demostrado que con una sola iteración, el resultado puede ser el deseado.

5.1.3. Eficiencia temporal

Uno de los aspectos importantes a analizar es la viabilidad temporal, es decir, estudiar si es viable la generación de código con estas metodologías y, además, hacer una comparación entre ellas.

Para analizar la eficiencia temporal, pues, hay que tener en cuenta que durante las llamadas a la API, ha habido varios problemas referentes a los tiempos de espera que ya se han comentado anteriormente. Por lo tanto, en las siguientes gráficas se observarán varios valores atípicos que no se deben tener en cuenta al valorar una metodología, aunque se dejará constancia de ello.

Una de las clases que mejor representa la proporción de tiempo entre metodologías es la clase *PriorityQueue*.

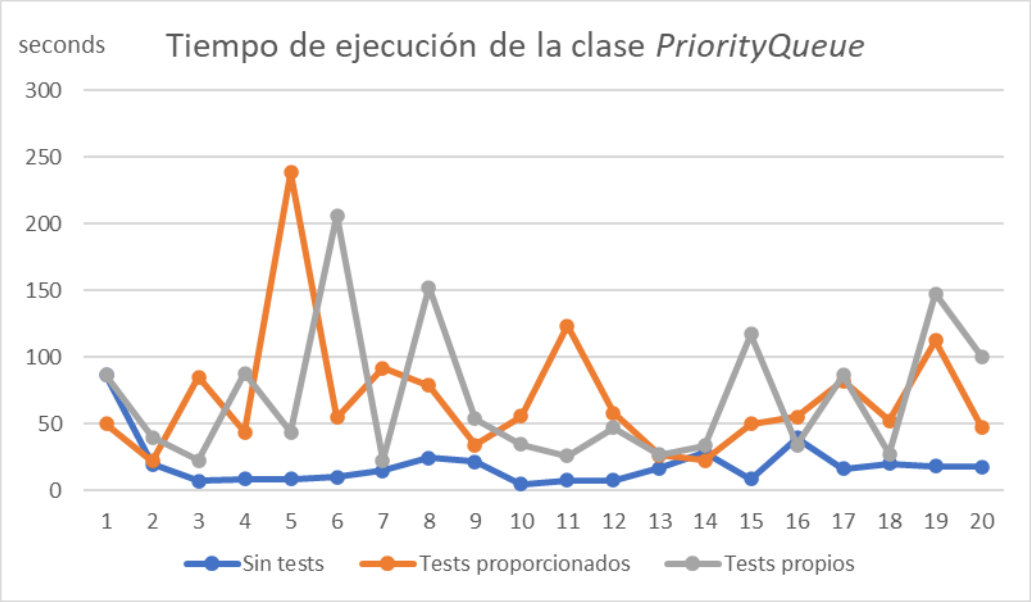


Figura 36. Gráfica que representa el tiempo utilizado para generar el código de la clase *PriorityQueue* según la metodología. Fuente: original.

A simple vista, ya se puede observar que la metodología que menos tiempo necesita y la más estable es la que no valida el código con tests. De hecho, es normal que sea así, ya que es la metodología que menos contexto envía y, en consecuencia, es la más eficiente temporalmente. Ahora bien, como ya se ha visto antes, esta es la que ofrece peores resultados.

Por lo tanto, al analizar las otras dos metodologías, el primer aspecto que llama la atención es la falta de regularidad que hay. Esto se debe a los *timeouts* que se han ido produciendo durante la ejecución de la generación de código. Lo que hacía que aumentara considerablemente el tiempo de espera. Ahora bien, no siempre se producían, en consecuencia, la gráfica generada es totalmente irregular.

A pesar de ello, cuando no se producían *timeouts*, el tiempo de espera es ligeramente superior al que se produce con la metodología que no utiliza tests. Entonces, debido a que ofrecen mejores resultados y la diferencia, eliminando los *outliers*, no es tan grande, se convierten en metodologías temporalmente válidas.

No obstante, para considerar una metodología no válida, se tendría que hablar en el orden de bastantes minutos, ya que hay que tener en cuenta que actualmente el código lo desarrolla una persona desde cero, tardando siempre más de los tiempos obtenidos en las pruebas. En cambio, si se utiliza esta herramienta, se reduce considerablemente el tiempo total de desarrollo porque se asume que el usuario terminará modificando el código generado por el LLM.

En cuanto a las otras clases, ha habido algunas que han sido mucho más estables que esta que acaba de comentarse. No obstante, estas gráficas no muestran una realidad, ya que el código generado con estas clases no tenía en ningún caso métodos con cuerpo definido. Por lo tanto, no tiene sentido comentarlas al detalle porque su valor es nulo.

Lo que sí merece ser comentado es la gráfica de la clase *Stack*.

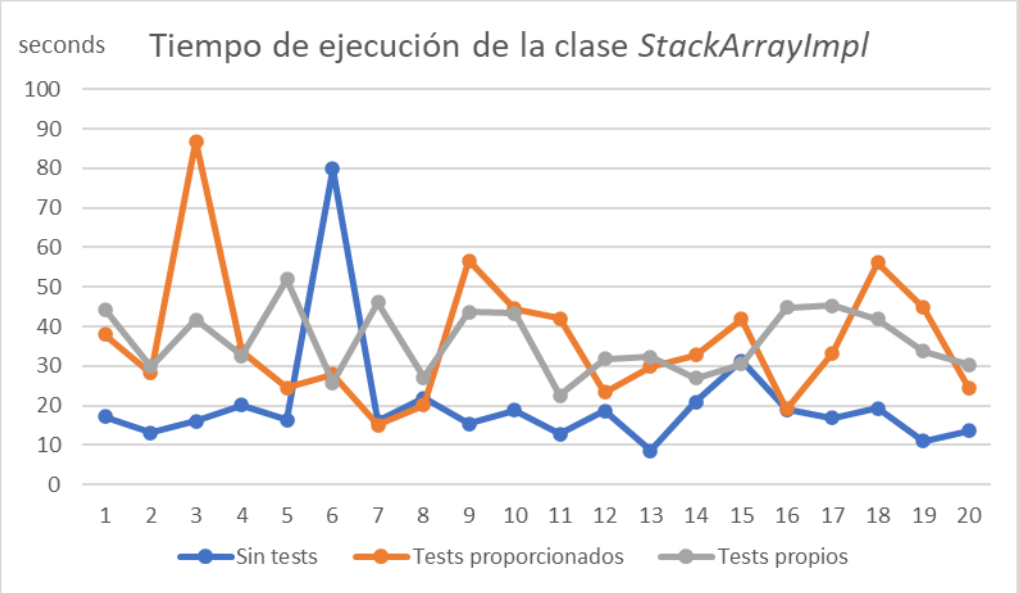


Figura 37. Gráfica que representa el tiempo utilizado para generar el código de la clase *StackArrayImpl* según la metodología. Fuente: original.

La parte interesante de esta gráfica, a diferencia de la anterior, es que la cantidad de tiempo utilizado es más similar entre metodologías. Esto es debido al hecho de que la clase generada tiene muchos menos métodos para desarrollar. Por lo tanto, de esta gráfica se llega a la conclusión de que, cuantos menos métodos se tengan que desarrollar, los tiempos de ejecución serán más similares entre metodologías. Sin embargo, la metodología que no genera tests, en líneas generales, sigue siendo la más eficiente temporalmente.

5.2. Ventajas y desventajas de cada metodología

En cuanto a las ventajas y desventajas de cada una de las metodologías, se han comentado de manera indirecta durante la memoria. Aun así, es importante hacer un resumen y tener en cuenta estos aspectos a la hora de elegir una u otra.

Sin tests

En relación a esta metodología, se ha observado que es la que ofrece una mejor eficiencia temporal. Adicionalmente, también es una metodología que se aplica sin necesidad de tener que generar ningún tipo de material manualmente, ya que no se envía nada más que el lenguaje natural definido por el usuario sobre lo que quiere generar.

Ahora bien, esta también es la que ofrece unos peores resultados frente a las otras dos. Donde, además, destaca que en múltiples ocasiones el código generado contenía métodos sin cuerpo. Además, los resultados tampoco mejoraban cuando el contexto enviado era más grande.

Consecuentemente, no es una metodología que sea recomendable para usar en un entorno profesional.

Con tests proporcionados

En cambio, esta metodología presenta muchas diferencias respecto a la que se acaba de comentar. Concretamente, una de las mejoras es que ofrece unos resultados bastante buenos, pero no suficientes para poder afirmar que es una metodología para seguir.

Asimismo, es la metodología que mejor ha tolerado los cambios que se han apreciado entre el cambio de versión del LLM.

Uno de los otros inconvenientes que presenta este caso es que requiere que exista una colección de tests previamente desarrollados manualmente. Por lo tanto, al tiempo de ejecución cuando se genera el código requerido, se le debe sumar el tiempo que se tarda en desarrollar los tests.

Así, pues, esta metodología, aunque no tiene unos resultados muy malos, se concluye que tampoco es la más adecuada, ya que tiene un coste temporal considerablemente alto.

Con tests propios

Finalmente, la última metodología presenta múltiples ventajas. La primera de ellas, que es la más importante, es que presenta los mejores resultados posibles. Sin embargo, a medida que aumenta el contexto que se envía al LLM, los resultados comienzan a empeorar.

Por otro lado, también ofrece ventajas porque no necesita que se cree ninguna colección de tests manualmente. La misma metodología se encarga de generarlos. Como resultado, no se tiene que sumar ningún otro tiempo como sí ocurría con la metodología anterior.

Como efectos negativos, se debe destacar que ha empeorado los resultados respecto a la última versión con la que se lanzaron las pruebas. A pesar de eso, no es la metodología que peores resultados ofrece.

Finalmente, también hay que decir que es la metodología con el peor coste temporal de todas. Tal como se ha comentado, debido a que la cantidad de tiempo de la que se estaba hablando, excluyendo los *outliers*, es de segundos, se considera que la metodología es viable temporalmente.

Por lo tanto, concluyendo este capítulo, según todos los análisis comparativos que se han llevado a cabo, se llega a la conclusión de que la mejor metodología que debería utilizarse según los estudios realizados, aunque no es perfecta, es la tercera, donde el propio LLM es quien valida el código con una colección de tests propia.

6. Despliegue del prototipo dentro de un IDE

Una vez analizadas las distintas metodologías de generación de código, se puede poner en práctica dentro de un IDE para que este pueda añadir el código generado directamente en el archivo de texto en el que se esté trabajando.

Para hacerlo, se ha partido de la premisa de que es importante mantener cierta privacidad sobre cómo funciona internamente el prototipo desarrollado. Es decir, el usuario final no necesita tener conocimiento sobre cuál es el *prompt* que se está utilizando para generar el código ni cómo se generan las llamadas a la API. Por lo tanto, se ha optado por ocultar esta información añadiendo un servidor en el cual se ejecute el prototipo. Precisamente por ese motivo se ha mencionado anteriormente que el prototipo debe tener la capacidad de ser multiplataforma, ya que de esta manera es mucho más fácil ponerlo en producción. Además, si el prototipo se ejecuta en un servidor, cualquier IDE podrá solicitarle el código de manera sencilla porque solo se tendrá que implementar el código referente al cliente, es decir, al IDE.

Así, lo primero que se ha hecho es registrar un servidor en AWS (Amazon Web Services) y se ha creado un entorno con Flask para poder habilitar una dirección que acepte consultas de tipo POST, donde se requiera un parámetro que indique lo que quiere generar el programador. De esta manera, como ya se ha avanzado, el servidor será quien realice la consulta a la API del LLM aplicando un *prompt* concreto y devolverá la respuesta al IDE que ha hecho la consulta.

El código que se ejecuta en el servidor es una versión reducida del prototipo que se puede encontrar en el Anexo IV, ya que solo requiere la metodología donde se han obtenido mejores resultados. Además, esta también permite generar tests de manera automatizada en función del código generado y del *prompt* recibido.

Consecuentemente, con el prototipo funcionando dentro de un servidor de manera generalizada, ya se puede desarrollar el código para cualquier IDE que lo permita para hacerle consultas.

6.1. Visual Studio Code

Visual Studio Code ofrece la posibilidad de añadir extensiones dentro de él, ya sean propias o de terceros. Por lo tanto, es el entorno ideal para poner a prueba el prototipo en un entorno profesional. Concretamente, para poder llevarlo a cabo, se ha creado una extensión con JavaScript que solo se permite ejecutar de manera local, es decir, no se ha publicado la extensión, ya que esta está en una fase experimental. A pesar de eso, con el objetivo de permitir entender mejor el funcionamiento de esta, se ha habilitado un repositorio público donde se puede consultar el código de la extensión. El repositorio se puede encontrar en el Anexo VII.

El objetivo de este trabajo no es mostrar todo el proceso por el cual se ha pasado para poder crear la extensión, pero a título de ejemplo se mostrará cuál es el funcionamiento de esta.

Concretamente, se ha añadido una nueva opción dentro del menú contextual que actúa como disparador de la extensión una vez se presiona el botón derecho del ratón si se ha seleccionado texto previamente.

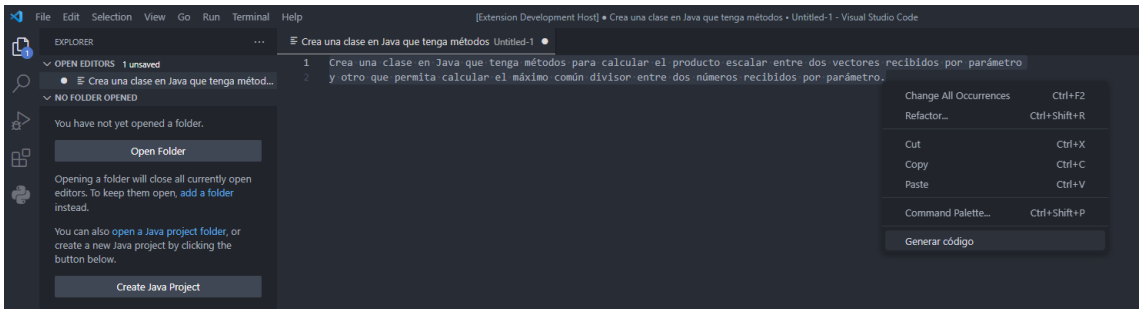


Figura 38. Captura de pantalla donde aparece la opción para generar el código solicitado con el prototipo desarrollado. Fuente: original.

Una vez se activa, se hace una llamada al servidor donde se ha desplegado el prototipo para poder iniciar todo el proceso. De esta manera, el servidor solicita al LLM el código utilizando el *prompt* con el cual se ha trabajado durante todo el estudio y devuelve la respuesta en formato XML.

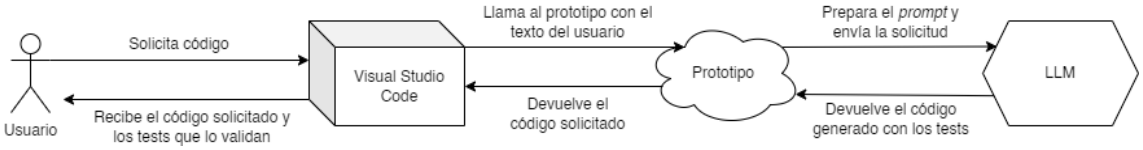


Figura 39. Modelo conceptual del sistema desarrollado. Fuente: original.

Seguidamente, el IDE añade el código generado dentro del archivo de texto desde donde se ha solicitado y muestra por el canal estándar de salida los tests generados con el resultado de su ejecución (hay que recordar que no se ejecutan realmente, el LLM no tiene esa capacidad, solo intenta prever su resultado).

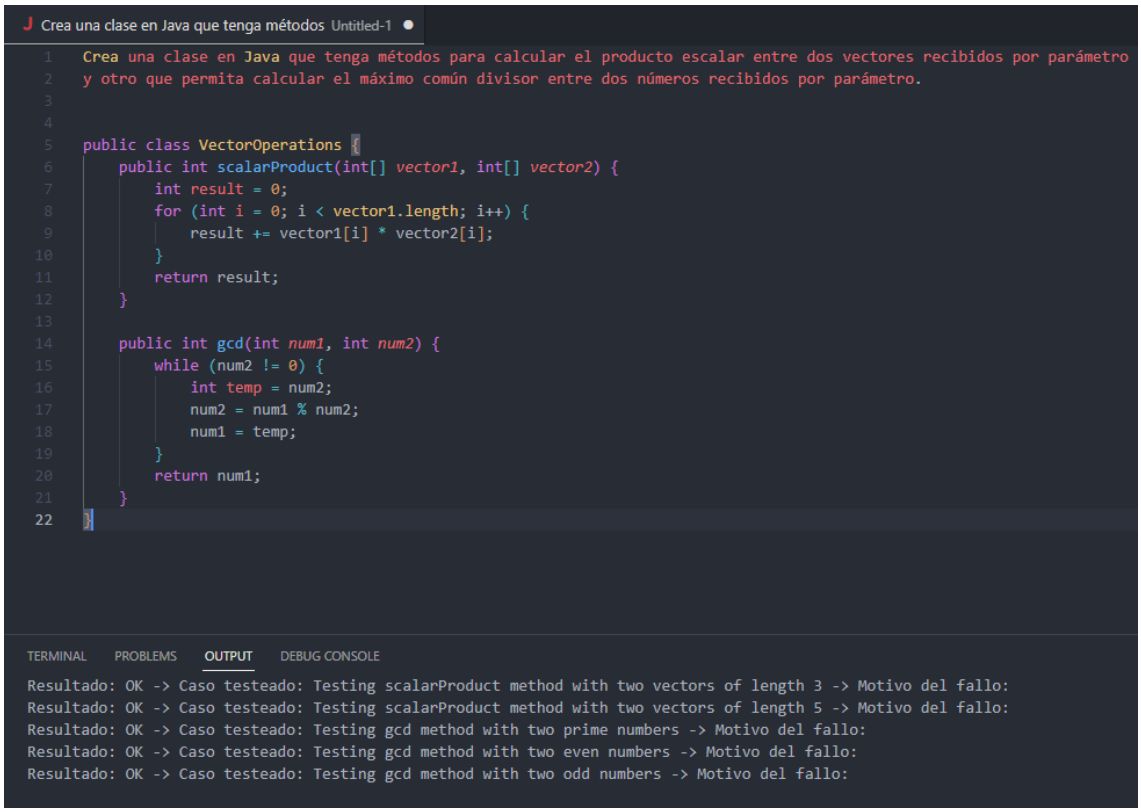


Figura 40. Captura de pantalla con el código añadido correctamente dentro del archivo de texto y el conjunto de tests con sus respectivos resultados. Fuente: original.

Finalmente, cabe destacar que el tiempo de espera depende completamente del código que se quiera generar. Pero, para el caso planteado de la Figura 38, el tiempo de espera no suele ser superior a 10 segundos. Así que, se considera que el tiempo de respuesta es considerablemente bueno teniendo en cuenta el tiempo que se tardaría en desarrollar el código desde cero.

Asimismo, se debe considerar que el prototipo desarrollado es una primera versión y que está en fase experimental. Por lo tanto, es bastante probable que surjan problemas inesperados, especialmente cuando el texto introducido por el usuario es muy largo. Estos deberían ser detectados y solucionados en futuras versiones del prototipo.

6.1.1. Retroalimentación con ChatGPT

En cuanto a la capacidad que ofrece el prototipo de comunicarse con el LLM utilizado, hay que decir que la comunicación se produce de dos maneras.

La primera ocurre en la misma generación del código, cuando el LLM proporciona una colección de tests creada por él mismo y ofrece información sobre su propio código. De esta manera, la persona que ha solicitado el código puede tener en cuenta las consideraciones que se han hecho y modificar, o no, el código generado.

La segunda tiene que ver con *active prompting*. Si bien es cierto que actualmente el prototipo no funciona usando una única cadena con múltiples respuestas que se van añadiendo, el usuario tiene la capacidad de escribir qué modificaciones se deben hacer y, con la selección de este nuevo texto y el código generado, puede volver a solicitar al prototipo que modifique el código generado.

En consecuencia, a pesar de no ofrecer mecanismos más eficientes, como el caso de una única cadena, la persona que quiere desarrollar el código tiene la capacidad de comunicarse directamente con el LLM mediante los tests que este genera.

7. Conclusiones y trabajos futuros

Como se ha podido detallar a lo largo de todo el trabajo, la aplicación de ciertas metodologías en la generación de código hace que el resultado pueda mejorar considerablemente. Ahora bien, dado el contexto que ha habido durante este estudio, no se han dado las condiciones para poder ofrecer la propiedad de ser un sistema determinista. Es decir, los resultados no siempre son los mismos y tampoco siempre son los esperados.

Como consecuencia, este modelo se puede usar como herramienta que mejora considerablemente la eficiencia temporal de una persona que desarrolla código. En ningún caso, se puede considerar hoy en día que un LLM pueda sustituir a un programador o programadora. Por lo tanto, también se debe afirmar que no se puede delegar la generación de código de manera no supervisada dada la situación actual.

Ahora bien, esto no implica que los avances que se han producido en este campo no sean muy interesantes. Tal como se ha visto en el capítulo 2 de este trabajo y se ha corroborado con los resultados obtenidos, el uso de los LLM en el futuro dentro de la generación de código debe ir en aumento. No solo lo indican la enorme cantidad de *papers* que se esmeran en este campo, sino también lo dicen los resultados que se han logrado en este mismo trabajo.

La aplicación de la metodología TDD enfocada en validar el código generado por un LLM muestra signos inequívocos de una mejora considerable respecto a lo que se obtiene sin ella. Aún así, la aplicación de esta metodología debe enfocarse de una manera diferente. En el momento de la redacción de este proyecto no había ciertas herramientas disponibles que ahora sí lo están. Por ejemplo, no se podían usar *plugins* vinculados a un LLM. Y, de hecho, esta posibilidad abre el camino que según varios estudios citados en esta memoria y las conclusiones a las que se ha llegado con este trabajo ya afirman, que es la necesidad de poder utilizar diversas herramientas controladas por un LLM.

Esta necesidad se ha visto reflejada en este mismo documento cuando se ha afirmado que se han encontrado clases generadas con métodos sin cuerpo donde el LLM entendía que los tests pasaban todos correctamente. Como resultado, si se dispusiera de un compilador al cual el LLM tuviera acceso, podría generar el código de los tests para poder ejecutarlos en un compilador real en vez de tener que predecir sus resultados.

Ya se avanzó en los primeros capítulos de este trabajo que un LLM no tiene la capacidad de compilar y ejecutar el código. La validación del código usando tests la hace a partir de lo que considera que ocurrirá según lo que ha leído y lo que ha redactado.

En este sentido, referenciando de nuevo al modelo *EvalPlus* del capítulo 2, sería muy interesante modularizar la generación de código con varias llamadas al LLM empleando diferentes modelos: uno para generar unos tests de calidad, que podría ser un modelo como *EvalPlus*, y el otro que permita generar el

código dados los tests que se han generado y, finalmente, poder compilar y ejecutar el código generado con los tests. Solo de esta manera se podría llegar a una situación donde se pueda afirmar que el sistema es determinista. Hecho que mejorará la confianza de la comunidad hacia estas herramientas.

Ahora bien, incluso pudiendo ofrecer resultados mucho mejores, se deben considerar varios aspectos que pueden ser un problema que provoque que el modelo propuesto no sea posible. Por ejemplo, se debería analizar el coste temporal que tendría un modelo así y ver si realmente es capaz de poder hacer solicitudes como la que se acaba de sugerir. Además, continuando con la línea de lo dicho anteriormente, los LLM carecen de una parte lógica que podría mejorar los resultados obtenidos. Si la tuviera, mejoraría la complejidad temporal del modelo de manera indirecta, ya que se llegaría a una situación donde los resultados son correctos de una manera más rápida.

Es decir, los LLM por sí mismos son capaces de proporcionar una ayuda considerable. Pero, si el objetivo que se persigue es dotar a un modelo de más independencia, se deben proporcionar las herramientas que se han mencionado, construyendo de esta manera un modelo mucho más complejo y sofisticado.

Adicionalmente, respecto al prototipo y la extensión desarrollados, deberían probarse en otros entornos, lenguajes y generando código del cual el LLM quizás no tiene tanto conocimiento. En este estudio se han solicitado clases en Java de ADT que son bastante conocidos. Por lo tanto, se debería ir más allá y pedirle código del cual no tiene tanto conocimiento.

Como ejemplo, se podría probar la extensión creada dentro de una aplicación profesional que requiera código no muy común. Haciendo uso de una metáfora con la universidad, se podría solicitar que generase una clase que represente un estudiante de la UOC. Al solicitarle eso, automáticamente requiere más contexto, ya que el LLM no puede saber que un usuario debe estar relacionado con aulas del campus o que debe tener un tutor o tutora asignado.

Consecuentemente, se introduce uno de los otros aspectos clave en el futuro: el número de *tokens* que se pueden enviar como contexto. Durante el estudio, el número máximo de *tokens* era 4.096. Actualmente, lo han aumentado hasta más de 16.000. Y, de hecho, en el futuro la tendencia es que este continúe aumentando. En consecuencia, existe un alto grado de dependencia sobre este número.

Si progresivamente el número va aumentado, puede llegar un punto donde el LLM se pueda introducir dentro de cualquier proyecto, metafóricamente hablando. Es decir, se puede dar una situación donde el LLM tenga conocimiento de todo el código de una aplicación o servicio. Dependiendo de lo grande que sea la aplicación, harán falta más o menos *tokens*. Pero, si en el futuro es viable proporcionar todo el contexto de una aplicación, los resultados mejorarían drásticamente y, en consecuencia, se estaría llegando a una situación donde el LLM podría llegar a ser capaz de generar código según las

necesidades que introduzca un programador o una programadora por muy grande que sea la aplicación.

En cualquier caso, esta situación mencionada es ciencia ficción actualmente y queda considerablemente lejos del alcance de este trabajo, pero sin duda es un escenario que seguramente querrá ser explorado en el futuro. Depende de cuán rápido se avance computacionalmente en este caso. La carrera ya empezó hace años.

“Desafiamos las fronteras de la tecnología y la ciencia con curiosidad y valentía hacia una singularidad que nos redefina.”

ChatGPT

8. Glosario

Active prompting: no existe una traducción exacta para este término, pero es una metodología que consiste en incentivar el pensamiento crítico de un LLM para mejorar su capacidad de razonamiento a través de interactuar con él.

ADT: *Abstract Data Type* en inglés. Es la representación de un modelo a partir de varios métodos que definen un comportamiento concreto. Por ejemplo, el de una lista enlazada.

API: *Application Programming Interface* en inglés. Software que permite establecer una comunicación entre dos programas.

AWS: Amazon Web Services. Es un servicio que ofrece la empresa Amazon donde se pueden alquilar servidores. Ofrece mucha escalabilidad y cuenta con productos muy concretos para tareas específicas como administración de bases de datos, aprendizaje computacional, etc.

Bug: error en inglés, especialmente relacionado con el mundo del software.

Chain-of-thought: cadena de pensamiento en inglés, es una metodología que permite mejorar la capacidad de razonamiento de un LLM mediante la generación de pasos intermedios que lo lleven a una respuesta correcta.

ChatGPT: inteligencia artificial preentrenada desarrollada por la empresa OpenAI en formato de chatbot que utiliza un LLM para dar respuesta a las consultas de los usuarios.

Context menu: es una interfaz que contiene varias opciones que aparece cuando el usuario interactúa con una aplicación o página web, normalmente con el botón derecho del ratón. Las opciones más comunes son las de copiar y pegar.

Flask: es un *framework* que permite la creación de aplicaciones web basadas en Python. Además, es eficiente a la hora de crear APIs, ya que ofrece mecanismos que generan el entorno necesario sin que el software tenga que preocuparse.

Framework: es un entorno de trabajo que incluye múltiples herramientas ya preparadas con el objetivo de minimizar y generalizar el trabajo de un desarrollador de código.

Getter: método de una clase utilizado para devolver el valor de un atributo.

GitHub Copilot: inteligencia artificial preentrenada desarrollada por las empresas GitHub y OpenAI que permite autocompletar código fuente en función del inicio de este utilizando un LLM.

IA: inteligencia artificial. Es la combinación de algoritmos utilizados de una determinada manera para ofrecer capacidades inteligentes como la tarea de clasificar, generar texto, etc.

IDE: *Integrated Development Environment*. Software que ofrece un editor de código fuente y facilidades para los programadores de software. Además, también incluye herramientas de compilación y de debug.

Java: es un lenguaje de programación orientado a objetos de alto nivel. Tiene la peculiaridad de poder ejecutarse en cualquier plataforma, es decir, es multiplataforma. Se suele utilizar para crear aplicaciones empresariales, aplicaciones móviles y en el desarrollo web.

Javadoc: es una herramienta que genera documentación en formato HTML de manera automática. En ella, se añaden los comentarios del código y toda la información referente a la clase: nombre, herencias, implementaciones, métodos, atributos, etc.

JavaScript: es un lenguaje de programación interpretado de alto nivel utilizado principalmente dentro del desarrollo web. Sin embargo, también se utiliza en otros campos y también existen entornos de trabajo basados en este lenguaje.

Langchain: biblioteca basada en Python utilizada para la creación de una cadena con el LLM.

Línea de comandos: es una interfaz que permite ejecutar órdenes por un usuario sobre un computador o un programa.

LLM: *Large Language Model* en inglés. Es un grafo entrenado con múltiples capas que permite generar texto a partir de lenguaje natural introducido por un usuario.

Outlier: valor de un conjunto de datos que difiere considerablemente de los demás. Normalmente suelen representar casos aislados producidos por errores en la medida.

Paper: la traducción directa al castellano es papel. Es una publicación académica producida por una o varias personas expertas en un campo de estudio concreto donde se presentan los resultados y las conclusiones de una investigación.

Plugin: es un componente que añade una funcionalidad concreta dentro de un software ya existente.

POST: es un método que permite enviar información cuando se realiza una consulta. Es decir, permite el envío de datos por parte de un usuario o un servidor hacia otro.

Python: es un lenguaje de programación interpretado de alto nivel especialmente utilizado en campos como la inteligencia artificial, aunque también se usa en otros campos como el desarrollo web.

Setter: método de una clase utilizado para asignar un valor a un atributo.

Spam: envío masivo de mensajes no deseados.

TDD: *Test Driven Development* en inglés. Es una metodología utilizada en el desarrollo de software donde, en una primera instancia, se desarrolla una colección de tests y, después, todo el desarrollo del código fuente está guiado por estas.

Timeout: es el escenario que ocurre cuando un proceso no finaliza dentro del tiempo esperado. Está creado con el objetivo de no tener procesos que nunca terminen o evitar el uso de recursos excesivos.

UML: *Unified Modeling Language* en inglés. Es un lenguaje unificado que persigue el hecho de poder representar el diseño conceptual de un sistema.

VPN: *Virtual Private Network* en inglés, es una red virtual privada que ofrece la capacidad de ocultar la dirección pública del usuario.

9. Bibliografía

[1] **K. Hu**, *"ChatGPT sets record for fastest-growing user base – analyst note"*, 2 de febrero de 2023, https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/ (última consulta: 31 de mayo de 2023).

[2] **A. Radford, K. Narasimhan, T. Salimans y I. Sutskever**, *"Improving Language Understanding by Generative Pre-Training"*, 2018, https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf (última consulta: 31 de mayo de 2023).

[3] **Y. Liu, T. Han, S. Ma, J. Zhang, Y. Yang, J. Tian, H. He, A. Li, M. He, Z. Liu, Z. Wu, D. Zhu, X. Li, N. Qiang, D. Shen, T. Liu y B. Ge**, *"Summary of ChatGPT/GPT-4 Research and Perspective Towards the Future of Large Language Models"*, 11 de mayo de 2023, https://arxiv.org/pdf/2304.01852.pdf (última consulta: 31 de mayo de 2023).

[4] **J. R. Rodríguez.** *"La gestió de projectes. Conceptes bàsics"*, PID_00247934, pp. 20-24, editorial UOC.

[5] **Y. Shen, K. Song, X. Tan, D. Li, W. Lu e Y. Zhuangb.** *"HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face"*, 25 de mayo de 2023, https://arxiv.org/pdf/2303.17580.pdf (última consulta: 10 de junio de 2023).

[6] **S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang y J. Gao.** *"Interactive Code Generation via Test-Driven User-Intent Formalization"*, 11 de agosto de 2022, https://arxiv.org/pdf/2208.05950.pdf (última consulta: 10 de junio de 2023).

[7] **J. Liu, C. S. Xia, Y. Wang y L. Zhang.** *"Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation"*, 12 de junio de 2023, https://arxiv.org/pdf/2305.01210.pdf (última consulta: 14 de junio de 2023).

[8] **A. Eleti, J. Harris y L. Kilpatrick, OpenAI.** *"Function calling and other API updates"*, 13 de junio de 2023, https://openai.com/blog/function-calling-and-other-api-updates (última consulta: 14 de junio de 2023).

[9] **C. Treude.** *"Navigating Complexity in Software Engineering: A Prototype for Comparing GPT-n Solutions"*, 28 de enero de 2023, https://arxiv.org/pdf/2301.12169.pdf (última consulta: 31 de mayo de 2023).

[10] **A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh y M. C. Desmarais.** *"GitHub Copilot AI pair programmer: Asset or Liability?"*, 14 de abril de 2023, https://arxiv.org/pdf/2206.15331.pdf (última consulta: 31 de mayo de 2023).

[11] **OpenAI.** *"Introducing ChatGPT Plus"*, 1 de febrero del 2023, https://openai.com/blog/chatgpt-plus (última consulta: 31 de mayo de 2023).

[12] **GitHub.** *"QuickStart for GitHub Copilot"*, sin fecha de publicación, https://docs.github.com/en/copilot/quickstart (última consulta: 31 de mayo de 2023).

[13] **OpenAI.** *"Code completion"*, sin fecha de publicación, https://platform.openai.com/docs/guides/code (última consulta: 31 de mayo de 2023).

[14] **M. Wilson-Thomas, G. Hogenson, D. Lee, C. Caldwell, K. Toliver, G. Warren, A. Weins y M. Nylander, Microsoft.** *"IntelliCode for Visual Studio overview"*, 27 de julio de 2022, https://learn.microsoft.com/en-us/visualstudio/intellicode/intellicode-visual-studio (última consulta: 31 de mayo de 2023).

[15] **S. Kang, J. Yoon y S. Yoo.** *"Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction"*, 22 de diciembre de 2022, https://arxiv.org/pdf/2209.11515.pdf (última consulta: 31 de mayo de 2023).

[16] **S. Diao, P. Wang, Y. Lin, R. Pan, X. Liu y T. Zhang.** *"Active Prompting with Chain-of-Thought for Large Language Models"*, 23 de mayo de 2023, https://arxiv.org/pdf/2302.12246.pdf (última consulta: 31 de mayo de 2023).

[17] **Langchain.** *"Introduction"*, sin fecha de publicación, https://python.langchain.com/docs/get_started/introduction.html (última consulta: 31 de mayo de 2023).

[18] **Sin autor.** *"11 Most In-Demand Programming Languages"*, 5 de enero de 2023, https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/ (última consulta: 10 de junio).

10. Anexos

11.1. Anexo I: Tabla de hitos inicial

Nombre del hito	Inicio	Fin
PAC 0		
Investigación *state of art*	01/03/23	06/03/23
Redacción propuesta	07/03/23	13/03/23
PAC 1		
Creación del diagrama de Gantt	14/03/23	20/03/23
Redacción del plan	21/03/23	28/03/23
PAC 2		
Preparación del entorno	29/03/23	29/03/23
Diseño del prototipo	30/03/23	01/04/23
Desarrollo del prototipo	03/04/23	05/04/23
Automatización con *javadoc* de la biblioteca de DED	06/04/23	11/04/23
Automatización con *javadoc* + tests de la biblioteca de DED	12/04/23	17/04/23
Automatización de creación de tests con *javadoc* de la biblioteca de DED	18/04/23	22/04/23
Análisis y comparación de los primeros resultados	24/04/23	29/04/23
Redacción del informe	01/05/23	03/05/23
PAC 3		
Implementación de la metodología *chain-of-thought* al prototipo	04/05/23	06/05/23
Implementación de la metodología *active prompting* al prototipo	08/05/23	10/05/23
Análisis y comparación de todos los resultados	11/05/23	15/05/23
Integración con Visual Studio Code	16/05/23	22/05/23
Añadir *feedback* sobre tests no superados	23/05/23	25/05/23
Redacción del informe	26/05/23	29/05/23
PAC 4		
Redacción de los contenidos	30/05/23	16/06/23
Revisión final	17/06/23	20/06/23

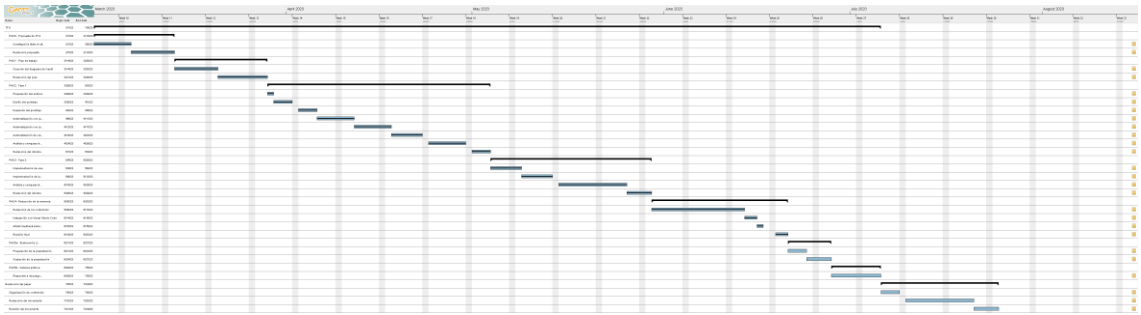
Nombre del hito	Inicio	Fin
PAC 5a		
Preparación de la presentación	21/06/23	23/06/23
Grabación de la presentación	24/06/23	27/06/23
PAC 5b		
Responder a las preguntas del tribunal	28/06/23	05/07/23
Redacción del *paper*		
Organización de contenidos	06/07/23	08/07/23
Redacción del documento	10/07/23	20/07/23
Revisión del documento	21/07/23	24/07/23

11.2. Anexo II: Tabla de hitos final

Nombre del hito	Inicio	Fin
PAC 0		
Investigación *state of art*	01/03/23	06/03/23
Redacción propuesta	07/03/23	13/03/23
PAC 1		
Creación del diagrama de Gantt	14/03/23	20/03/23
Redacción del plan	21/03/23	28/03/23
PAC 2		
Preparación del entorno	29/03/23	29/03/23
Diseño del prototipo	30/03/23	01/04/23
Desarrollo del prototipo	03/04/23	05/04/23
Automatización con *javadoc* de la biblioteca de DED	06/04/23	11/04/23
Automatización con *javadoc* + tests de la biblioteca de DED	12/04/23	17/04/23
Automatización de creación de tests con *javadoc* de la biblioteca de DED	18/04/23	22/04/23
Análisis y comparación de los primeros resultados	24/04/23	29/04/23
Redacción del informe	01/05/23	03/05/23
PAC 3		
Implementación de una cadena para poder enviar ejemplos de código	04/05/23	08/05/23
Implementación de la metodología *active prompting* al prototipo	09/05/23	13/05/23
Análisis y comparación de todos los resultados	15/05/23	25/05/23
Redacción del informe	26/05/23	29/05/23

Nombre del hito	Inicio	Fin
PAC 4		
Redacción de los contenidos	30/05/23	13/06/23
Integración con Visual Studio Code	14/06/23	15/06/23
Añadir *feedback* sobre tests no superados	16/06/23	16/06/23
Revisión final	17/06/23	20/06/23
PAC 5a		
Preparación de la presentación	21/06/23	23/06/23
Grabación de la presentación	24/06/23	27/06/23
PAC 5b		
Responder a las preguntas del tribunal	28/06/23	05/07/23
Redacción del paper		
Organización de contenidos	06/07/23	08/07/23
Redacción del documento	10/07/23	20/07/23
Revisión del documento	21/07/23	24/07/23

11.3. Anexo III. Diagrama de Gantt del trabajo



Dado que la imagen tiene unas dimensiones superiores al ancho de este documento, se ha subido la imagen a Google Drive públicamente. La imagen puede ser consultada desde el siguiente enlace: imagen del diagrama de Gantt.

Adicionalmente, se ha subido la versión web que permite exportar el programa GanttProject en un sitio web personal para poder visualizarla en este formato: diagrama de Gantt en formato HTML.

11.4. Anexo IV. Repositorio del código fuente del prototipo

El repositorio donde reside el código del prototipo se ha subido a GitHub de manera pública. Se puede acceder mediante el siguiente enlace: repositorio del prototipo.

11.5. Anexo V. Diagrama UML del prototipo

Dado que el diagrama UML del prototipo tiene unas dimensiones más grandes que el ancho del documento, se ha subido una captura de este en el siguiente enlace: diagrama UML del prototipo.

11.6. Anexo VI. Repositorio de gráficas

Para consultar todas las gráficas generadas se puede acceder mediante el siguiente enlace: repositorio de gráficas.

11.7. Anexo VII. Repositorio de la extensión de Visual Studio Code

El repositorio donde reside el código de la extensión para Visual Studio Code se ha subido a GitHub de manera pública. Se puede acceder mediante el siguiente enlace: repositorio de la extensión.